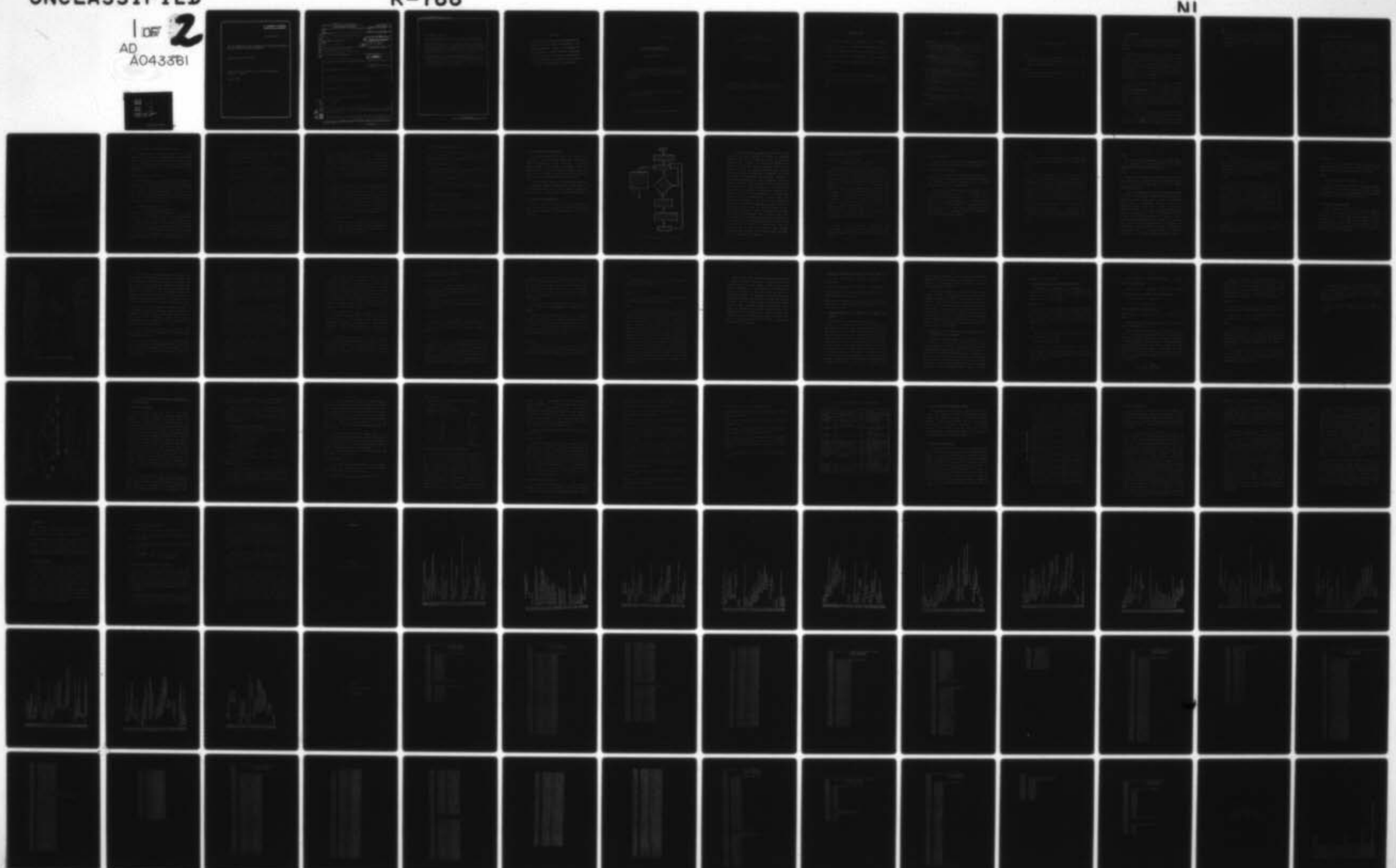
AD-A043 361 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORD SCIENCE LAB F/G 9/2
AN ALGORITHM FOR MINIMIZING PROGRAM LOGIC ARRAY REAL--ETC(U)
APR 77 A.G. SOONG DAAB07-72-C-0259
UNCLASSIFIED R-766 NL

1 OF 2
AD
A043361



SIFIED

1 OF 2

AD
A043361

1·0 2·8 2·5
3·15 2·2
3·5 2·0
1·1
1·8
1·25 1·4 1·6

NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

AD-A043 361

AN ALGORITHM FOR MINIMIZING PROGRAMMABLE LOGIC ARRAY REALIZATIONS

Alphonso Gar-Yau Soong

University of Illinois at Urbana-Champaign
Urbana, Illinois

April 1977

AD A043361

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
AN ALGORITHM FOR MINIMIZING PROGRAMMABLE LOGIC ARRAY REALIZATIONS,

5. TYPE OF REPORT & PERIOD COVERED
Technical Report,

6. PERFORMING ORG. REPORT NUMBER
R-766, UILU-ENG-77-2213

7. AUTHOR(s)
Alphonso Gar-Yau Soong

8. CONTRACT OR GRANT NUMBER(s)
DAAB-07-72-C-0259,
NSF-MCS-72-03488 A01

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Joint Services Electronics Program

12. REPORT DATE
April 1977

13. NUMBER OF PAGES
92

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

PLA Minimizations

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Due to the increasing use of PLAs (Programmable Logic Arrays) in logic design, an efficient algorithm which performs multiple-output AND-OR logic minimization is desired.

Quine-McCluskey (QM) logic minimization has been known for sometime.[1] It can provide AND-OR structures with a minimum number of gates, and secondarily, gate inputs. Unfortunately, the QM method is generally practical only for small numbers of inputs (under 10) and outputs (under 6). The enormous size

DD FORM 1473, 1 JAN 73 EDI

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

of the multiple output prime implicant covering table used in this method for large problems makes it too expensive to be implemented.

Other known algorithms either have similar complexity or provide only "good", but not necessarily optimum solutions. Therefore, a new AND-OR minimization algorithm for logic problems with up to 16 inputs and 8 outputs (standard limitations of PLAs available at present) is needed. The algorithm should be particularly effective for problems which require no more than 40 to 50 product terms in an optimum realization.

In this report, such an algorithm is formulated which strives to achieve an AND-OR realization with the smallest number of AND gates, without regard to the number of input connections per AND gate or the number of input connections per OR gate. This goal derives directly from the fact that only the number of product terms (AND gates) per PLA is limited by the PLA structure. The basic structure of this algorithm was originally suggested to the author by E. S. Davidson.

NOTICE

UILU-ENG 77-2213

AN ALGORITHM FOR MINIMIZING
PROGRAMMABLE LOGIC ARRAY REALIZATIONS

by

Alphonso Gar-Yau Soong

This work was supported in part by the Joint Services Electronics Program (U. S. Army, U. S. Navy and U. S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the National Science Foundation under Grant MCS 73-03488 A01.

AN ALGORITHM FOR MINIMIZING PROGRAMMABLE LOGIC ARRAY REALIZATIONS

BY

ALPHONSO GAR-YAU SOONG

B.S., University of Illinois, 1975

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor E. S. Davidson

Urbana, Illinois

ACKNOWLEDGEMENT

I wish to express my gratitude to my advisors, Professor Edward S. Davidson and Professor Jacob A. Abraham, for suggesting this thesis topic and for their invaluable guidance and friendship. Their continued patience and support in supervising this work are much appreciated.

I would also like to thank my wife, Tina, for her encouragement and help to put this report into its final form.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1. Introduction

Due to the increasing use of PLAs (Programmable Logic Arrays) in logic design, an efficient algorithm which performs multiple-output AND-OR logic minimization is desired.

Quine-McCluskey (QM) logic minimization has been known for some time.[1] It can provide AND-OR structures with a minimum number of gates, and secondarily, gate inputs. Unfortunately, the QM method is generally practical only for small numbers of inputs (under 10) and outputs (under 6). The enormous size of the multiple output prime implicant covering table used in this method for large problems makes it too expensive to be implemented.

Other known algorithms either have similar complexity or provide only "good", but not necessarily optimum solutions. ~~Therefore~~ A new AND-OR minimization algorithm for logic problems with up to 16 inputs and 8 outputs (standard limitations of PLAs available at present) is needed. The algorithm should be particularly effective for problems which require no more than 40 to 50 product terms in an optimum realization.

In this report, such an algorithm is formulated which strives to achieve an AND-OR realization with the smallest number of AND gates, without regard to the number of input connections per AND gate or the number of input connections

per OR gate. This goal derives directly from the fact that only the number of product terms (AND gates) per PLA is limited by the PLA structure. The basic structure of this algorithm was originally suggested to the author by E. S. Davidson.

2. Description of the algorithm

The AND-OR minimization algorithm discussed here is a branch-and-bound algorithm which makes a series of locally optimum decisions using the concepts of switching theory to derive a first solution. After finding the first solution, it backtracks to consider alternative decisions, modifying gate inputs and successively improving the solution. If run to completion, the algorithm finds a minimum gate solution. At each point the maximum improvement obtainable from continuing to run the algorithm is known.

The algorithm starts by choosing, heuristically, one minterm (1-cell) from one function of the given set of output functions. The smallest cube is found which covers this minterm and all its neighbour minterms in the selected function. Note that this cube may cover some Ø-cells of that function. All the minterms inside this cube are said to be covered or potentially covered. This cube is also potentially useful for other functions in the set which contains the selected minterm. Minterms of such functions which are inside the cube are also said to be potentially covered. The cube is then entered into an (initially empty) list called LISTA. Then another minterm which is not covered or potentially covered by cubes in LISTA is chosen and the process is repeated until each minterm of each output function is covered or potentially covered by some cube in LISTA. The resulting set of cubes in LISTA can be

transformed into feasible realizations of the output functions by shrinking the cubes, adding minterm variables to their corresponding product expressions, and adding further cubes when minterms become uncovered, until all 1-cells of the given set of output functions are covered and all the Ø-cells in the set of cubes in LISTA are eliminated. Different choices of variables for shrinking cubes correspond to different possible realizations of the output functions. A branch-and-bound method is used so that all possible realizations can be examined implicitly. That is, instead of finding all feasible realizations, the algorithm only continues to develop a class of solutions if some solution in the class has a chance of improving the best solution yet found. The last solution found before the algorithm halts is an optimum realization.

A formal description of the algorithm is presented in the following sections.

2.1 Preliminary Definitions

Definition (Term)

A term is a logical product of one or more variables some of which may be complemented and some of which may be enclosed in parentheses, ().

Definition (Maximum and Minimum Cube of a Term)

The maximum cube of a term is the set of all cells covered by the term if all parenthesized variables were deleted from the term. The minimum cube of a term is the set of all cells covered by the term if all parenthesized variables are replaced by the same variables without parentheses.

Definition (Partial Solution)

A partial solution is a set of terms each of which is assigned to a single function in the given set of functions to be realized. The minimum cube of each term must include only 1-cells of its assigned functions. For each term, if any (single) parenthesized variable was deleted from the term, the minimum cube of the resulting term would include only 1-cells of its assigned function.

For example, consider the function

$$f(w,x,y,z) = \sum (0,1,4,6,7,13,15)$$

The term w'(x')y'(z') could be assigned to f in a partial solution since its minimum cube, (0), and the minimum cubes of w'(x')y', (0,1), and w'y'(z'), (0,4), contain only 1-cells of f. Note that it is not required that the maximum cube of the term, (0,1,4,5) corresponding to w'y', contain only 1-cells of f and in fact in this case, (5) is a 0-cell of f. For this term assigned to f, w' and y' may not be parenthesized since (8) and (2) are not 1-cells of f.

Definition (Cover and Potentially Cover)

A term covers its minimum cube. A term potentially covers its maximum cube. For example: w'(x')y'(z') covers w'x'y'z' and potentially covers w'y'.

Definition (Useful and Potentially Useful)

A term is useful for a function if the cells covered by its maximum cube are all 1-cells of the function. A term is potentially useful for a function if the cells covered by its minimum cube are 1-cells of the function.

In the previous example, we might wish to know what terms are useful for function f and cover cell (Ø). Of course w'x'y'z' is such a term. From the restrictions on parenthesized variables in terms assigned to f, we know that w'x'y' and w'y'z' are such terms as well. These terms result from deleting a single parenthesized variable and deleting all other parentheses. Terms resulting from deleting two or more parenthesized variables are such terms if they may be assigned to f in a partial solution. In this example, w'y' is not such a term since (5) is not a 1-cell of f.

In the algorithm to follow, terms are created for the purpose of covering a 1-cell of a function, the minterm representing the selected 1-cell is constructed and all variables in the minterm which may be parenthesized, while preserving assignability to f, are written in parentheses.

As the algorithm proceeds, parenthesized variables and parentheses are deleted from terms. When a parenthesized variable is deleted from a term, thereby expanding its minimum cube, parentheses around other variables may have to be deleted from the term, thereby shrinking its maximum cube. In our example, if either parenthesized variable in w'(x')y'(z') is deleted, the remaining pair of parentheses must be deleted as well, to preserve the assignment of the term to f.

Useful terms remain useful no matter which parentheses or parenthesized variables are deleted. Potentially useful terms which are not useful become useful if certain parentheses are deleted. They may become not potentially useful and not useful if certain parenthesized variables are deleted. Note that there is no difference between useful and potentially useful if the term does not have any parenthesized variables. In that case, the maximum cube of the term is the same as the minimum cube of the term.

Definition (Uncovered Cell)

A 1-cell of a function is said to be uncovered in a partial solution if it is neither covered nor potentially covered by any term in the partial solution that is potentially useful for that function.

Definition (Transformation of a Term)

A term, T1, is a transformation of a term, T2, if T1 can be obtained from T2 by deleting some pairs of parentheses and some set of parenthesized variables from T2.

Definition (Intermediate Solution)

An intermediate solution is a partial solution in which no 1-cell of any function is uncovered.

Definition (Feasible Solution)

An intermediate solution is a feasible solution if no term in the intermediate solution contains any parenthesized variables.

Definition (Potentially Redundant)

A term in an intermediate solution is said to be potentially redundant if the deletion of the term from the intermediate solution would not generate any uncovered cells for any output functions.

Definition (Table of Usefulness)

The table of usefulness is a table for partial solution which shows for each function the terms which are useful and those which are potentially useful.

2.2 Basic process of the algorithm

All output functions to be realized are input to the algorithm as sum of products expressions. The number of distinct product terms in these expressions is an upper bound, UPBOUND, on the number of product terms in the optimum solution. Any other feasible solutions with the same or higher number of product terms are no better than the orginal input and therefore are of no interest.

The algorithm for finding an optimum AND-OR realization of a multiple output function consists of two phases. In the description below, ()'s is used to denote "parentheses" and ()-variable is used to denote "parenthesized variable."

2.2.1 Phase 1 of the algorithm

Phase 1 begins with a partial solution containing no terms and produces an intermediate solution by adding terms to the partial solution. A flow chart of Phase 1 is shown in Figure 1.

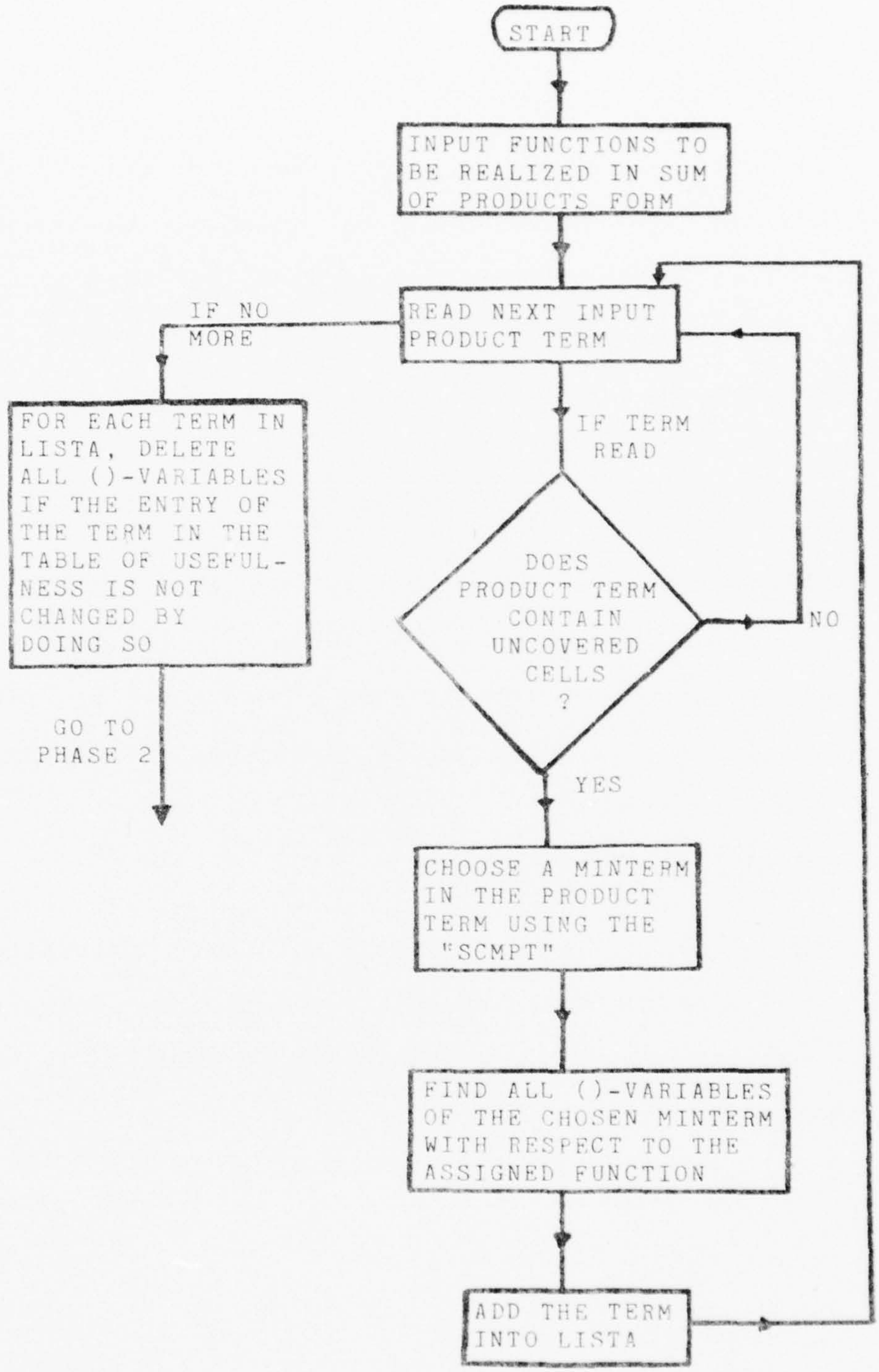


FIGURE 1 : FLOW CHART OF PHASE 1 OF ALGORITHM

It begins by choosing a product term from the input expression for some function and selects an uncovered minterm (1-cell) in this product term using the Selection Criterion for a Minterm in a Product Term (SCMPT), which will be discussed later. For our purposes, SCMPT may be assumed to select an arbitrary uncovered cell. Then the directions in which this minterm can be expanded (to cover two minterms of the function) are determined. Variables of the minterm corresponding to these expandable directions become ()-variables in the term and the term is added to the set of terms of the partial solution. By repeating this process until no minterms of any function are uncovered, the orginal set of terms is augmented to an intermediate solution with the characteristics outlined above. Just before the exit of Phase 1 to Phase 2 the intermediate solution may be modified by expanding some terms. A term is expanded if and only if for each function for which the term is potentially useful or useful, its maximum cube contains only 1-cells of that function. Then its entry in the table of usefulness is not changed by deleting all ()-variables. In this case, all ()-variables are deleted so that the term may cover all the cells of its maximum cube. This step allows the cube to grow to its maximum extent without precluding consideration of any optimum solution and simply makes the algorithm more efficient.

It is important to note that the maximum cubes of the terms might contain some 0-cells.

For example, consider the function

$$f(w,x,y) = w'y + w'xy' = \sum(1,2,3)$$

Let the minterm chosen from the first input term be w'xy, i.e. cell (3), then the term obtained is w'(x)(y), which covers (3) but potentially covers (0,1,2,3). Note that minterm (0) is not in f. The significance of the term w'(x)(y) is that every sum of products form for f must cover minterm (3) with a term which is a transformation of w'(x)(y). However, not all transformations of w'(x)(y) need be allowed. Each time a ()-variable is deleted, the other ()-variables must be re-evaluated to see if their ()'s must be removed. In this case the allowable transformations of w'(x)(y) with no ()-variables are w'x, w'y and w'xy. The remaining possible transformation, w', is not allowed. Since there are no uncovered cells of f once the term w'(x)(y) is in the list, w'(x)(y) is an intermediate solution and no further minterms are selected.

Theorem 1 states an important property of the intermediate solution produced by Phase 1. In order to prove it, however, we need some definitions and preliminary results.

Definition (Implicant)

An implicant of a function f is a product term (with no ()-variables) which covers only 1-cells of the function.

Definition (Proper Transform)

For a LISTA term, T, generated for minterm M of function f (i.e. generated just after minterm M of function f is selected), the proper transforms of T are those transforms of T which are implicants of f.

Note that the proper transforms of T, generated for M of f, all cover the minimum cube of T. After Phase 1, i.e. before Phase 2, the minimum cube of T is M (unless T is expanded by the last step of Phase 1). If expansion of T occurs, let T represent the term before expansion and LISTA represent the set of terms in the intermediate solution before expansion. The expansion step will be justified at the end of the discussion of Phase 2.

Lemma 1:

If T is a LISTA term generated for M of f during Phase 1, every implicant of f which covers M is a proper transform of T.

Proof:

Let I be an implicant of f which covers M. Any variable which appears complemented or uncomplemented in I must appear complemented or uncomplemented, respectively, in M and hence likewise in T, otherwise I would not cover M. For any variable which does not appear in I at all, the cell adjacent to M found by complementing that variable in the expression for M must be a 1-cell of f, since it is covered by I. Hence T must contain that variable as a ()-variable.

Thus there is a transformation of T which equals I, namely that transformation of T which deletes all ()-variables which do not appear in I and deletes all other ()'s. This transformation is a proper transformation since it is an implicant of f and contains no ()-variables. Q.E.D.

Lemma 2:

If T is a LISTA term generated for M of f during Phase 1, no implicant of f which covers M is a proper transformation of any LISTA term except T.

Proof:

Suppose T1 is a LISTA term generated for M1 of g and some implicant, I, of f which covers M is a transformation, t1, of T1. We will show that t1 is not a proper transformation of T1.

Since all transformations of T1 cover M1 and t1 equals I, then I must cover M1. Thus M1 must be a 1-cell of f. Therefore T1, whose minimum cube is M1, is potentially useful for f. Furthermore, since t1 covers M, T1 potentially covers M. Now M of f could not have been selected in Phase 1 if M1 of g had been selected first, since T1 in LISTA would not leave M of f uncovered. Thus M1 of g must have been selected after M of f. However, since I must be a proper transform of T, T potentially covers M1. Now T must not be potentially useful for g, otherwise M1 of g could not be selected after T is in LISTA. Therefore M, the minimum cube of T, must be a Ø-cell of g. Since t1 covers M, t1 is not a proper transformation of T1. Q.E.D.

Theorem 1 :

Given LiSTA produced by Phase 1 for a set of functions and an arbitrary sum of products expression for each function in the set, there is some proper transformation of each LISTA term which appears in the sum of products expressions and these terms are distinct.

Proof:

Each term in LISTA is generated for some minterm of some function. Let T in LISTA be generated for M of f. In any sum of products expression for f, there must be at least one term which covers M. Let I be an arbitrary one of these terms. By Lemma 1, I is a proper transformation of T. By Lemma 2, I is not a proper transformation of any other LISTA term. Similarly there is some proper transformation of each LISTA term which equals some expression term and each of these expression terms is logically distinct from the others. Q.E.D.

By Theorem 1, the terms of every set of sum of products expressions for a set of functions can be constructed as an appropriate transformation for each LISTA term produced by Phase 1 and possibly some added terms.

Corollary 1 :

The cardinality of LISTA after Phase 1 is less than or equal to the number of terms in any set of sum of products expressions for the set of functions input to Phase 1.

Proof:

Follows immediately from Theorem 1. Q.E.D.

Therefore, at the end of Phase 1 a lower bound, LBOUND = cardinality of LISTA, and an upper bound, UPBOUND = number of distinct terms in the input expressions, are established for the number of terms in an optimum solution.

2.2.2 Phase 2 of the algorithm

Phase 2 examines the intermediate solution obtained from Phase 1 and proceeds to find a succession of feasible solutions, each with fewer terms than the previous feasible solution. Phase 2 will halt and upon halting, the last feasible solution found has the minimum number of terms among all feasible solutions of the problem. The flow chart of Phase 2 is presented in Figure 2.

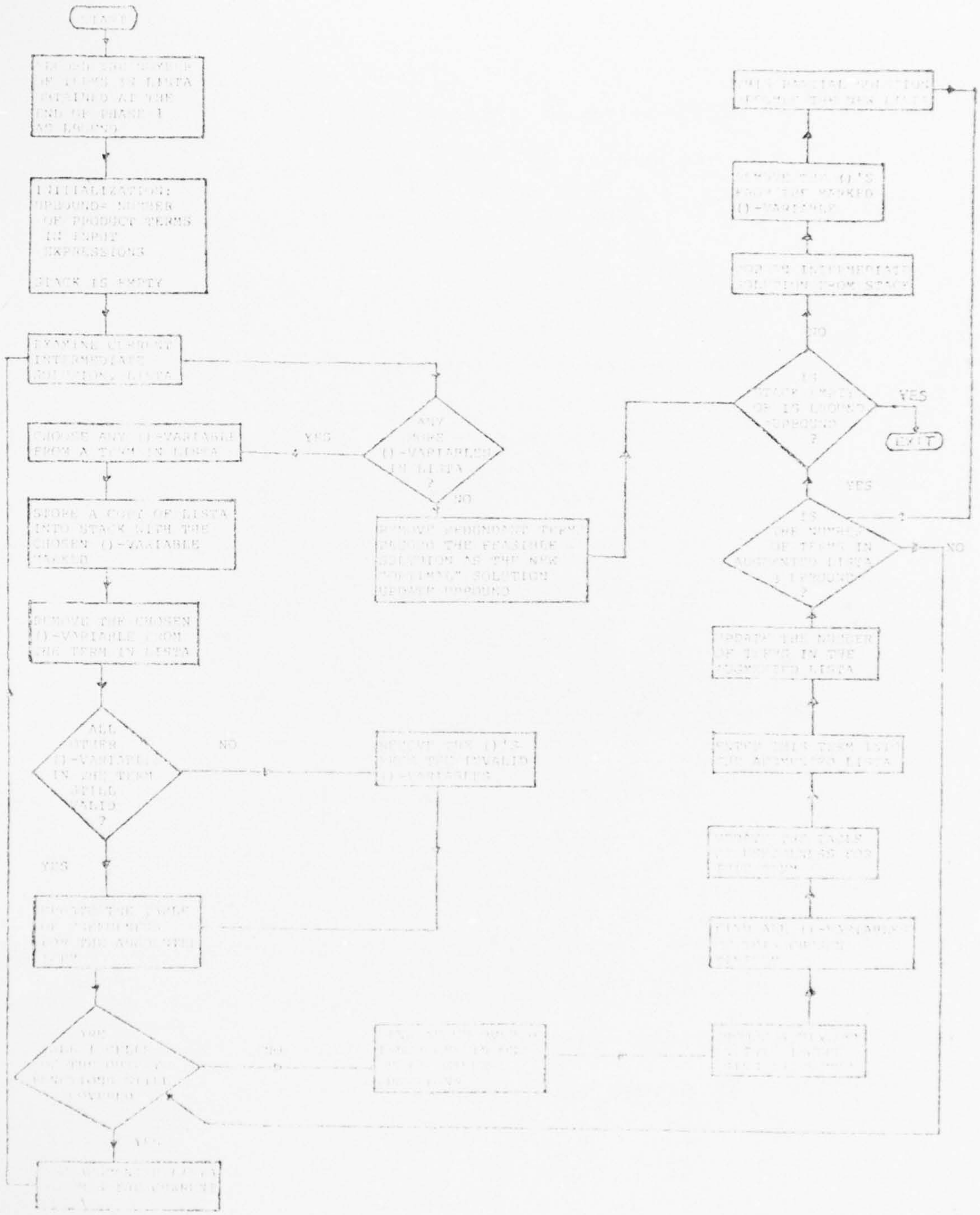


FIGURE 2 : FLOW CHART OF PHASE 2 OF ALGORITHM

A ()-variable is arbitrarily chosen from a term in the intermediate solution and a 2-way branch is performed. One of the branches corresponds to removing the ()-variable from the term. This is equivalent to retaining the maximum cube of the term but doubling the size of the minimum cube with respect to the ()-variable. The other branch corresponds to removing the ()'s from the variable. This is equivalent to reducing the maximum cube of the term by a half with respect to the ()-variable while the minimum cube remains the same.

Both branches have the effect of reducing the number of ()-variables in the term, a procedure which will eventually transform the chosen term into a legitimate product term of the given output functions. Since both branches (transformations) may uncover some 1-cells, subroutines of Phase 1 must be called to check :

(1) If the rest of the ()-variables in the term are still valid. If not, ()'s may have to be removed from some ()-variables of the term.

(2) If all 1-cells of the given set of output functions are still potentially covered by the terms in each of the two transformed lists of LISTA. If not, appropriate terms must be added to the two lists respectively to cover the uncovered 1-cells.

Step (1) arises when the minimum cube of a term is expanded, i.e. when a ()-variable is deleted. In order to insure that any ()-variable of this term may be deleted without making the transformation of the term cover any 0-cells of the function, ()'s must be removed from those ()-variables which do not correspond to an expandable direction of the new minimum cube (even though they did correspond to an expandable direction for the previous minimum cube).

Step (2) arises when a ()-variable is deleted since the minimum cube becomes larger and the term may become potentially useful for a smaller set of functions. Cells which the term potentially covers in functions for which the term is no longer potentially useful may become uncovered. Step (2) also arises when ()'s are deleted since the maximum cube becomes smaller and cells which are no longer potentially covered by this term in functions for which this term is potentially useful may become uncovered.

After this checking process, two new intermediate solutions are formed. Only one of these two intermediate solutions (the one with the ()-variable deleted) is selected to be used as the new input to Phase 2. The other one is stored in a stack called STACK for later backtracking. Then the whole process is repeated with Phase 2 focusing on the new intermediate solution.

Phase 2 thus contains a routine which is repeated iteratively until the intermediate solution under consideration has no more ()-variables in it, that is, until a feasible solution is found. All redundant terms in this feasible solution are deleted. Then the feasible solution is stored as the current "optimal" solution and replaces the old "optimal" solution. (The realization used in the input to the algorithm is the initial "optimal" solution.) The number of distinct product terms in this feasible solution becomes the new upper bound, UPBOUND.

If at any time in this process the number of terms in the intermediate solution under consideration is greater than or equal to the current upper bound, UPBOUND, that intermediate solution is discarded and the algorithm backtracks. A new intermediate solution is obtained from STACK to be used as the input to the iterative routine of Phase 2.

The algorithm stops when either a feasible solution consisting of only LBOUND distinct product terms is found or when all the intermediate solutions in STACK have been processed by Phase 2. The last "optimal" solution recorded is an optimum realization for the given set of output functions.

We now prove the optimality of the final solution produced by Phase 2 before halting.

Definition (Reachable from LISTA)

A solution is called reachable from LISTA if it contains a set of |LISTA| distinct terms each of which is a proper transformation of a distinct LISTA term (and possibly some other added terms).

Note that by Theorem 1, all solutions are reachable from the LISTA produced by Phase 1 (before expansion of selected T terms).

Lemma 3 :

All solutions reachable from the LISTA input to the iterative routine of Phase 2 are reachable from at least one of the two LISTA's output from the iterative routine of Phase 2.

Proof :

Consider the term and the ()-variable selected by the iterative routine. All solutions reachable from the input LISTA contain a term which is a proper transformation of that term. Furthermore this solution term is distinct from those corresponding to other LISTA terms. Hence that proper transformation must be a proper transformation of the selected term with the selected ()-variable either missing

or appearing without ()'s. The proper transformation must thus be a proper transformation of the term which replaces the selected term in one of the two output LISTA's. This statement is valid whether or not other ()'s are deleted from the term since other ()'s are deleted only to remove improper transformations. They remove no proper transformations.

No other terms in LISTA are modified by the iterative routine. Hence their correspondence to solution terms is unchanged.

Further terms may be added to LISTA by the iterative routine. However, these are added in a manner similar to Phase 1 only to cover uncovered cells of functions. It can be shown by an argument similar to that of Lemmas 1 and 2 and Theorem 1 that the added terms are necessary and do not affect reachability of solutions. Q.E.D.

Corrolary 2 :

The number of terms in the LISTA output from the iterative routine of Phase 2 is a lower bound on the number of terms in any feasible solution reachable from that LISTA.

Proof :

Follows immediately from Theorem 1 and the proof of Lemma 3 by finite induction. Q.E.D.

Theorem 2 :

The last solution produced by Phase 2 before halting is an optimum (minimum number of terms) solution.

Proof :

All solution are reachable from the LISTA produced by Phase 1, by Theorem 1. By Lemma 3, no reachable solutions are eliminated by the branching in Phase 2. By Corrolary 2 and the structure of the backtracking in Phase 2, all feasible solutions are fully developed except those with UPBOUND or more terms. UPBOUND is monotonically decreased during the run of Phase 2, but a solution is produced with UPBOUND terms for each value of UPBOUND. Thus the only solutions not produced by the algorithm have the same number of terms or more terms than some feasible solution produced by the algorithm. Furthermore, the last solution produced by the algorithm before halting has the fewest terms of any feasible solution produced by the algorithm. Thus any other solution to the problem has the same number of gates or more gates than the last feasible solution produced by the algorithm. Q.E.D.

There are two steps, the term expansion step at the end of Phase 1 and the casting out redundancy step when a feasible solution is found in Phase 2, which might require further explanation. Since expanded terms contain only 1-cells of functions for which the terms are useful or potentially useful, the expansion does not eliminate any solutions with fewer terms than the minimum-term solution reachable from the modified LISTA. This property follows from the prime implicant theorem of switching theory. Casting out redundant terms in Phase 2 serves only to reduce UPBOUND when possible and does not preclude reaching any solutions with fewer gates than UPBOUND. These steps thus only make the algorithm more efficient without jeopardizing finding an optimum solution.

3. Heuristics and special techniques used in the algorithm

In this minimization algorithm, heuristics are introduced in :

(i) the Selecting Criterion of a Minterm in a Product Term

(ii) selecting the branching priority with respect to the arbitrarily chosen ()-variable.

Also a special technique is used to solve the problem of deciding if a specific product term is covered by a given set of product terms.

3.1 Selecting Criterion of a Minterm in a Product Term (SCMPT)

When examining the input product terms in Phase 1, a minterm must be chosen from some input product terms to be the nucleus of a product term. Then the direction in which this minterm can be expanded is examined to determine the ()-variables in this minterm. Heuristically, the minterm which is covered by the least number of distinct prime implicants of the output functions should have the highest priority. This is because the maximum cubes formed by these minterms would be very 'tight', that is they will cover very few Ø-cells. This will reduce the work required to be done in Phase 2 and will tend to make Phase 2 converge to the optimum solution faster. Yet this process requires knowing how many 'Ø' neighbours a minterm has. A tedious and

time-consuming procedure has to be used to obtain this information and this process is impractical. A less efficient but very simple selecting criterion is chosen in this minimization algorithm.

In the program, terms in the problem description are scanned in order. For each term which contains one or more minterms uncovered by LISTA, one uncovered minterm is selected. Some minterm which is covered by only one input product term is chosen with highest priority because the maximum cube of this minterm would tend not to include too many 0-cells. If such a minterm cannot be found in the input product term, then a minterm is chosen arbitrarily from the input product term to serve as the nucleus of that product term. As a result the lower bound obtained at the end of Phase 1 is fairly tight.

3.2 Selection of the branching priority with respect to the arbitrarily chosen ()-variable

In Phase 2, a two-way branch may be performed on any ()-variable in LISTA until no ()-variables remain, i.e. a feasible solution is reached. Heuristically, the branch corresponding to deleting the ()-variable from the term is a better choice because this directly implies a reduction in the input load of the term and also an increase in the covering power of the minimum cube of the term. In the case when there is more than one optimum solution, this selection would tend to find the one with a smaller number of input

connections to the AND gates.

3.3 Special technique for the "covering" problem

The problem to be solved here is to determine if a product term P is covered by a set of N product terms namely, X1, X2, X3, ..., XN. This problem can be transformed into a simpler problem.

Theorem 3 :

A product term P is covered by a set of N product terms Xi (i=1,...,N) iff the union of the product terms Yi (i=1,...,N) is equal to '1', where Yi is the product term resulting from removing all the literals of P from the product term P.Xi.

Proof:

By the inclusion property: $P \subseteq X1 + X2 + X3 + \ldots + XN$ iff $P=P.(X1 + X2 + \ldots + XN)$

By the distributive law:

$$P.(X1 + X2 + \ldots + XN) = P.X1 + P.X2 + \ldots + P.XN$$

Since $P.Xi \subseteq P$ for all i, therefore there exists a Yi such that Yi and P are literal-disjoint and P.Xi = P.Yi for all i=1,...,N . Then

$$P.X1 + P.X2 + \ldots + P.XN = P.Y1 + P.Y2 + \ldots + P.YN$$

By the transitive law:

$$P \subseteq X1 + X2 + \ldots + XN \text{ iff } P = P.(Y1 + Y2 + \ldots + YN)$$

But since Yi (i=1,...,N) and P are literal-disjoint,

$$P = P.(Y1 + Y2 + \ldots + YN) \text{ iff}$$

$$Y1 + Y2 + \ldots + YN = '1' .$$

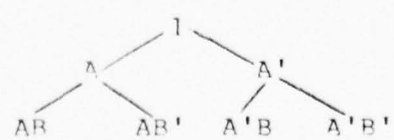
Again by applying the transitive law,

$$P \subseteq X1 + X2 + \ldots + XN \text{ iff } Y1 + Y2 + \ldots + YN = '1' .$$

Q.E.D.

The reduced problem can be easily solved by the tree method described below.

Each node of the tree represents a product term. The node at the top level is the product term 1. Each node is branched out to form two new nodes. One branch corresponds to adding (ANDing) one more literal in the uncomplemented form to the product term; the other to adding the same literal in the complemented form. A complete tree is formed when no more literals are available for branching from any node. For example, the complete tree of literals (A,B) is as shown below:



A node N1 is defined as a successor of node N2 if N1 can be obtained from N2 by adding literals to N2. In other words, N1 can be reached by branching out from N2. A node of the tree is said to be covered if it is covered by some product term Yi, i=1,...,N . If all successors of a node are covered, then the node is also said to be covered.

Theorem 4 :

Let the product terms Yi, i=1,...,N be product terms among which appear M variables namely, Aj, j=1,...,M.

Y1 + Y2 + ... + YN = '1' iff each node of the tree of variables (Aj, j=1,...,M) is covered.

Proof:

It is obvious that Y1 + Y2 + ... + YN = '1' iff all possible product terms of variables (A1,A2,...,AM) are covered by Y1 + Y2 + ... + YN. Since the tree of variables (A1,A2,...,AM) explicitly represents all possible product terms formed by variables (A1,A2,...,AM) the theorem is proved. Q.E.D.

It is important to note that it may not be necessary to examine all nodes of the tree explicitly, since once a node is covered by a product term all its successors are also covered by the same product term.

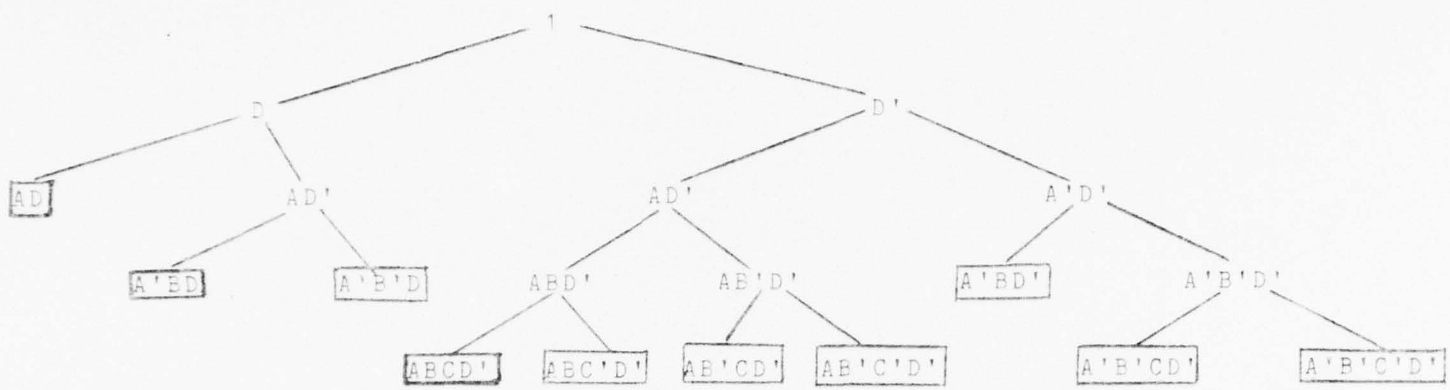
The problem of testing if a node N (a product term) is covered by another product term Yj can be easily solved because it is equivalent to testing if the set of literals appearing in the product term Yj is a subset of the set of literals appearing in N.

Example "TEST TAUTOLOGY", as shown in Figure 3, illustrates this tree method of solving the "tautology" problem.

Example "Test Tautology" :

Test : C'D' + A'B + A'B'D + B'CD' + AD + ABCD' = 1 ?

The tree of variables (A,B,C,D) is constructed as shown below :



Any node enclosed in implies that that node is covered.

Since all the nodes of the tree are covered (either covered by some product term or have all successors covered by product terms), therefore the sum of the product terms :

C'D' + A'B + A'B'D + B'CD' + AD + ABCD'

is indeed equal to 1.

FIGURE 3 : EXAMPLE "TEST TAUTOLOGY"

4. Description of the program that finds an optimum sum of products network

4.1 Scope of the program

The program 'MINI' has been coded for the DEC-10 computer in SAIL (Stanford Artificial Intelligence Language). It derives an optimum combinational AND-OR (sum of products) realization of a set of output functions. The algorithm is based on the branch-and-bound method discussed in the previous sections. Because of the nature of the branch-and-bound method, the first feasible solution found is not necessarily an optimum realization of the functions. After generating the first feasible solution, the program searches for better feasible solutions by backtracking. Each time the program finds a feasible solution whose total number of product terms is not greater than the previous "optimal" feasible solution, the program prints out the feasible solution and the number of product terms in the feasible solution. Eventually the program enumerates all feasible solutions (implicitly) and the last feasible solution found is an optimum solution for the output functions.

The user may specify the initial upper bound on the number of product terms to a value he thinks is reasonably low, in order to prune off some non-optimum networks, which would otherwise be generated by the program. This may reduce the computation time. If this number is not

specified the program will take the number of distinct product terms in the input as the initial upper bound.

The source code of the entire program occupies 53 blocks of storage on the DEC-10. The object code occupies 60 blocks of storage. The compilation time of the program is approximately 8 seconds. A listing of the program can be found in Appendix A.

4.2 Set-up of input data to the program

The input data is stored in an input file called 'DATA0'. 'DATA0' contains three types of lines.

(i) <problem-parameter>

(ii) <function-specifications>

(iii) <input-terminator>

<Problem-parameter>

The first line of DATA0 specifies NV, the number of variables in the functions.

The second line of DATA0 specifies NF, the number of output functions.

<Function-specification>

The set of functions is entered in some sum of products form. Each line corresponds to a product term in the input expressions. Each line is coded as a character string of '0', '1' and '-'. The character strings are each of length equal to the sum of NV and NF.

The first NF characters specify which output functions contain the product term associated with this line in their expressions. This part of the string constitutes an entry in a Table of Uselness for the input terms. A 'Ø' in the i-th position, where $1 \leqslant i \leqslant NF$, denotes that the product term is not in the expression for the i-th output function, and a '1' denotes that the product term is in the expressions. This part of each line contains a single '1' and NF-1 'Ø's.

The last NV characters specify a product term. A 'Ø' in the j-th position, where $1+NF \leqslant j \leqslant NF+NV$, implies that the (j - NF)-th variable in the complemented form appears in the product term and a '1' implies that the variable appears in the uncomplemented form in the product term. A '-' in the j-th position implies that the (j - NF)-th variable does not appear in the product term.

<Input-terminator>

The last line of DATAØ is the character string '999'. This tells the program that the end of function specification and input has been reached.

Example 'INPUT' illustrates a DATAØ input file.

Example 'INPUT':

Consider two output functions of three variables.

F1(W,X,Y) = WXY' + W'Y

F2(W,X,Y) = W'Y + X'Y' + WX

Then DATAØ is set up as follows:

LINE	INPUT	COMMENTS
1.	3	NV
2.	2	NF
3.	1Ø11Ø	WXY' of F1
4.	1ØØ-1	W'Y of F1
5.	Ø1Ø-1	W'Y of F2
6.	Ø1-ØØ	X'Y' of F2
7.	Ø111-	WX of F2
8.	999	Input Terminator

4.3 Interpretation of program output strings

The output of the program is essentially the same as the input except that more characters are involved in the strings. The first NF characters display an entry in the Table of Usefulness for the solution found. typically only feasible solutions need be printed, but the output format applies as well to intermediate or partial solutions which are also printed in the present version. The last NV characters represent the corresponding product term in LISTA. For the first NF characters, a '7' in the i-th position means that the term, specified by the last NV characters of the string, is potentially useful for the i-th

output function. The meanings of 'Ø' and '1' are that the term is not useful or is useful, respectively, for the function. The meanings of 'Ø', '1' and '-' in the last NV characters of the string are the same as in the input. A '2' in the j-th position, where 1+NF < j < NF+NV, means that the (j - NF)-th variable is in the complemented form and is also a ( )-variable. A '3' is the same as a '2' except that the variable is in the uncomplemented form.

Example 'OUTPUT' illustrates the interpretation of an output string.

Example 'OUTPUT':

Consider a problem of three output functions of five variables, namely V,W,X,Y,Z. Let the output functions be F1,F2 and F3. An output string Ø1712-3Ø is interpreted as: term V(W')(Y)Z' is useful for F2 and potentially useful for F3. Note that if the last NV characters in an output string do not contain characters '2' or '3', then any '7' in the first NF characters is equivalent to a '1'. To reduce term output loading, a minimum set of '7' terms should be selected to cover each function. Each term must be a '1' term for at least one function.

4.4 Subroutines of the program

The program 'MINI' consists of a main procedure PROGRAM, which is the outer-most block, and twelve subroutines, CHOX, COMPARE, INLIST, INTERB, INTERF, PAREN,

REDUN, SORT, TAUTOLOGY, UNION, UPTAB, UPTAB2, and I/O subroutines which are provided with SAIL compiler. Major functions of the subroutines are listed below.

CHOX: Choose a cell in a product term to be the nucleus of the product term using the SCMPT.

COMPARE: Compare the product terms of two input character strings to see if they are equal.

INLIST: Check if all the cells of an input product term are covered by the existing terms in the current partial solution. If not, create one and check for proper ( )'s around variables.

INTERB: Find the intersection of a product term with all the product terms in the partial solution that are potentially useful to the same set of functions for which the product term is useful or potentially useful.

INTERF: Find the intersection of a product term with the sum of all the input product terms of a function for which the product term is useful.

PAREN: To determine which variables of a minimum cube can be ( )-variables.

REDUN: Find any product terms or terms in the intermediate solution which are redundant.

SORT: Sort the input product terms in the order of

increasing number of '-'s in the product term.

TAUTOLOGY: To check if the union of a set of product terms is equal to logical '1' .

UNION: Check if a product term is covered by a given list of product terms.

UPTAB: Update the Table of Usefulness when an input product term is useful for more than one output function.

UPTAB2: Update the Table of Usefulness for any product term. This includes updating the usefulness and potential usefulness of a term or product term for any output functions.

A cross-reference table of the subroutines is presented in Table 1.

TABLE 1 : CROSS-REFERENCE TABLE OF SUBROUTINES

Procedure	Procedures it calls	Procedures calling it
CHOX	INTERF, UNION	INLIST, MAIN PROGRAM
COMPARE	-	MAIN PROGRAM
INLIST	CHOX, PAREN, UPTAB2 INTERB, UNION	MAIN PROGRAM
INTERB	-	INLIST, REDUN, MAIN PROGRAM
INTERF	-	CHOX, PAREN, UPTAB2, MAIN PROGRAM
PAREN	INTERF, UNION	INLIST, MAIN PROGRAM
REDUN	INTERB, UNION	MAIN PROGRAM
SORT	-	MAIN PROGRAM
TAUTOLOGY	-	UNION
UNION	TAUTOLOGY	INLIST, CHOX, REDUN, PAREN, UPTAB2, MAIN PROGRAM
UPTAB	-	MAIN PROGRAM
UPTAB2	INTERF, UNION	INLIST, MAIN PROGRAM

5. Test Problems and analysis of results

The program "MINI" was used to find realizations of several test single and multiple output switching problems. Results are recorded in Table 2. For test problems, detailed function specifications and the corresponding solutions found are listed in Appendix B. A detailed listing of the program output for test problem 1, the 7-segment decoder, is included in Appendix C for reference as a sample output of the program "MINI".

5.1 Results of test problems

As indicated in Table 2, optimal realizations were found for some of the test problems and the program halted. However, the other test problems were stopped from executing further because either the solutions obtained were good enough (the number of gates in the best solutions obtained thus far was close to the corresponding lower bound), or heuristically, it seemed that the program would require a large execution time to improve the best solution found for a problem at the point it was stopped. However, it must be noted that if all these test problems that were stopped are allowed to run to completion, optimal solutions would be found.

TABLE 2 : RESULTS OF TEST PROBLEMS

TEST PROBLEM	NUMBER OF INPUT VARIABLES	NUMBER OF OUTPUT FUNCTIONS	NUMBER OF DISTINCT PRODUCT TERMS IN PROBLEM	LOWER BOUND OBTAINED AT THE END OF PHASE 1	NUMBER OF ()-VAR AT THE END OF PHASE1	TIME TAKEN TO FINISH PHASE 1	NUMBER OF GATES IN SOLUTION OBTAINED	DEPTH OF TREE FOR SOLUTION	NUMBER OF INTER-MEDIATE SOLUTIONS PROCESSED	TOTAL EXECUTION TIME
1 7 SEGMENT DECODER	4	7	15	7	3	7 SEC	9 8 (OPTIMAL)	9	9 140	[illegible] SEC 1 MIN [illegible] SEC
2 [illegible] SHIFTER	13	3	[illegible]	68	5	1 MIN 45 SEC	68 (OPTIMAL)	5	5	2 MIN 40 SEC
3 [illegible] BINARY	3	7	58	25	52	3 MIN 2 SEC	42 40 (STOPPED)	65 55	65 123 [illegible]	5 MIN 10 SEC 13 MIN [illegible] SEC [illegible] MIN
4 [illegible] BINARY MULTIPLIER	5	5	49	16	37	21 SEC	37 35 35 33 (STOPPED)	66 63 62 60	65 69 75 823 1017	5 MIN 6 SEC 5 MIN [illegible] SEC 5 MIN 12 SEC 54 MIN 10 SEC 1 HR 17 MIN
5 [illegible] DECODER	12	7	66	52	79	11 MIN 4 SEC	- (STOPPED)	- -	- 812	3 HR
6 [illegible] SHIFT/ROTATE DECODER	14	8	127	116	29	3 MIN 12 SEC	124 (STOPPED)	21	21 72	9 MIN 37 SEC 30 MIN
7 SPECIAL COUNTER	5	5	31	14	22	37 SEC	21 16 (STOPPED)	31 29	31 43 120	54 SEC 1 MIN 12 SEC 23 MIN
8 F(W,X,Y,Z) = (W'X'Y'Z' + WXYZ)'	4	1	14	4	12	1 SEC	6 5 4 (OPTIMAL)	12 9 7	12 24 33	1.5 SEC 2.2 SEC 3 SEC
9 SPECIAL DECODER	8	1	48	8	18	7 SEC	12 (STOPPED)	10 -	10 132	20 SEC 5 MIN 30 SEC
10 [illegible]	5	1	16	4	13	1.5 SEC	5 5 (OPTIMAL)	8	8 30	6 SEC 13 SEC

5.2 Analysis of results

It is obvious that the efficiency of the algorithm is highly problem dependent. The total execution time required to solve a switching problem depends on the number of input variables in the problem, the number of output functions, and most important, the structure of the prime implicants of the problem.

As the number of input variables and the number of output functions in a problem increase, the problem space becomes larger and therefore the execution time required to solve the problem is generally increased. However, due to the nature of the branch-and-bound method, the most important factor governing the amount of execution time required to solve a problem is the structure of the prime implicants in the set of output functions in the problem. If the prime implicants are densely gathered, that is, 1-cells typically are members of many prime implicants, then a large number of ()-variables in the intermediate solution may remain at the end of Phase 1 of the algorithm and there are many seemingly good alternative ways of removing them. In order to find an optimum realization, the algorithm must then do many forward branching and backtracking steps which consume much execution time. This fact can be observed from results of test problems 3, 4, and 5. Therefore, if the test problem has a large number of ()-variables at the end of Phase 1 of the algorithm, the execution time that the

program takes to solve the problem tends to be large.

However, if there is a big difference in size between two problems, the program may take less time to solve the "smaller" problem even if the number of ()-variables at the end of Phase 1 for the "smaller" problem is larger than that of the "larger" problem. For example, consider the results of the test problems 8 and 10 vs those of test problems 1 and 2.

Also note that although test problem 7 is a much "smaller" problem than test problem 2, yet the time needed to solve test problem 2 is much shorter than that needed to solve test problem 7. This is because the prime implicants of problem 2 are scattered and there is very little sharing of terms between output functions. Therefore the last step of Phase 1 is able to cut the number of ()-variables in LISTA from 81 down to 5. So very little branching and backtracking needs to be done to solve the problem. On the other hand, the prime implicants of test problem 7 is densely gathered. There is a lot of sharing of 1-cells between the output functions. This results in a lot of branching and backtracking in Phase 2. Therefore when the program "MINI" was used to solve test problem 7, it ran for 20 minutes and still did not halt and had to be stopped. This illustrates that the structure of the prime implicants of a problem is a dominating factor on the performance of the algorithm.

Another important fact is that if the initial input specification of a test problem is very good, i.e. near optimum, and the prime implicants of the problem are densely gathered, as in the case of test problem 5, the program may run for an extremely long time and still not be able to find any better solution than the original input. This is because the input may already be an optimum realization. Yet, the program would still have to try branching on all those intermediate solutions with a fewer number of gates than the initial expression while searching for an optimum solution, or verify that the input is an optimum solution. This procedure results in a large amount of execution time especially when the number of ()-variables at the end of Phase 1 for the problem is large. This property is illustrated by the results of test problem 5.

Finally, the entry under the heading : "Depth of Tree of Solution" may require further explanation. The entry in this column for each test problem indicates that if starting from the intermediate solution LISTA, obtained from Phase 1, all branchings (either deleting parentheses from ()-variables or deleting ()-variables) have been selected correctly, the solution can be reached from the initial LISTA in exactly the recorded number of branchings.

6. Conclusions

The algorithm discussed above is an entirely new approach to solving the problem of minimizing two level AND-OR multi-output switching function realizations.

The efficiency of the algorithm is highly problem dependent. The algorithm works particularly well for problems with scattered prime implicants. Thus it is hard to derive any specific correlation between the execution time needed to solve a problem and the size of the problem.

6.1 Use of the algorithm

An attractive point about this algorithm is that after a brief execution time it can find a reasonably tight lower bound on the number of gates required to realize a given set of output functions. Therefore, whenever a satisfactory solution (not necessarily optimum) is found with a number of gates close to the lower bound, the program may be stopped if optimality is not the main objective of using this algorithm. In particular, if the objective of using this algorithm is to minimize the number of PLAs required to realize a given set of output functions, the following criterion can be used to stop the program.

Criterion for stopping the program :

Let the number of AND gates avaliable per PLA be P.

Let the lower bound found at the end of Phase 1 of the algorithm be LBOUND.

Let the number of gates in a feasible solution found by the algorithm be N.

If $\left\lceil \frac{LBOUND}{P} \right\rceil = \left\lceil \frac{N}{P} \right\rceil$ then STOP PROGRAM

else CONTINUE.

6.2 Possible improvements of the algorithm

It is unfortunate that there have not been many programs written for minimizing multi-output functions. Therefore not enough data can be obtained to measure the relative performance of the program "MINI". However, improvement in execution time can definitely be obtained if the program is coded in assembly language and run on some large computer.

Further improvement of the algorithm may be made if some better heuristics can be found to be added in the SCMPT or better and quicker methods to solve the "covering" problem are found.

In the present version, the program scans lexicographically in each intermediate solution and selects the first ()-variable found and branching is done with respect to this selected variable. Thus loading of variables will tend to be higher for variables that are lexicographically near the end. Also for other reasons, some good heuristics for ()-variable selection should be added.

Also in the optimum solutions found, 1-cells of an output function may be covered by more than one product term. A small cover problem could be solved to get a minimum set of '7' terms for each function to reduce redundancy.

Finally, the algorithm was designed with the prime objective of minimizing totally specified multi-output switching functions. However, modifications can be made such that the algorithm may be used to solve problems with "DON'T CARES" in inputs. This modification would treat "DON'T CARES" as 1-cells except that "DON'T CARE" minterms are never treated as uncovered cells and would never be selected as the nucleus of any LISTA term in Phase 1 of the algorithm. Then the problem can be solved in the usual manner.

APPENDICES

Appendix A

Listing of the program "MINI"

```
BEGIN "MINI";
 COMMENT PHASE0: TO PRODUCE A LIST OF DISTINCT PRODUCT TERMS
                         SUCH THAT THE ORDER IS IN DECREASING NUMBER
                         OF LITERALS ;
  INTEGER NV,NF,NPT,NPTX;
  COMMENT NV = NUMBER OF VARIABLES
          NF = NUMBER OF FUNCTIONS
          NPT = NUMBER OF PRODUCT TERMS ;
  STRING ARRAY INPT(1::250);
     COMMENT FIRST NF CHARACTERS REPRESENT WHICH FUNCTION DOES
             THE PRODUCT TERM BELONG, THE FOLLOWING NV CHARACTERS REPRESENT
             THE PRODUCT TERM ;
     COMMENT DEFINITION OF THE SYMBOLS USED:
        1 = X
        0 - X'
        - - LITERAL ABSENT
        2 = (X)
        3 - (X') ;
 COMMENT START OF PROGRAM ******* ;
  INTEGER CHAN1,CHAN2,LINE,BRCHAR,EOF,COUNT1 ; COMMENT I/O ;
  STRING BUF,TEMP  ;
  INTEGER ARRAY NUMLIT(1::250);
  BOOLEAN FLAG,FLAG1,FLAG2,EXIST;
  INTEGER PTRP,PTRF,L,I,J,LBOUND;



 PROCEDURE COMPARE(VALUE STRING A,B);
 COMMENT *** COMPARE TWO PRODUCT TERMS TO CHECK IF THEY ARE EQUAL ***;
   BEGIN INTEGER J;
      FLAG2_TRUE; J_PTRP;
      WHILE (FLAG2) AND (J LEQ L) DO
         IF ( A(J FOR 1) NEQ B(J FOR 1)) THEN FLAG2_FALSE
                                         ELSE J_J+1;
    END;



 COMMENT  *** UPDATING THE TABLE WHEN AN INPUT PRODUCT TERM IS USED IN MORE THAN ONE OUTPUT FUNCTION *** ;
 PROCEDURE UPTAB(VALUE STRING A,B);
   BEGIN INTEGER DONE;
     J_PTRF; DONE_0;
     WHILE (J LEQ NF) AND (DONE=0) DO
        IF ( A(J FOR 1) = "0" ) AND ( B(J FOR 1) = "1" ) THEN
                        DONE_1 ELSE J_J+1;
   END;



 PROCEDURE SORT;
 COMMENT : BY COUNTING THE # OF LITERALS IN EACH P.T. FIRST;
 BEGIN "SORT SUBROUTINE" FOR I_1 STEP 1 UNTIL NPTX DO
        BEGIN NUMLIT(I)_0; BUF_INPT(I);
              FOR J_PTRP STEP 1 UNTIL L DO
                 IF (BUF (J FOR 1) NEQ "-") THEN NUMLIT(I)_NUMLIT(I)+1;
        END;
 COMMENT BEGIN SORTING ;
 FOR I_1 STEP 1 UNTIL NPTX-1 DO
   BEGIN J_1;
         WHILE (J LEQ NPTX-1) DO
```

```
            BEGIN IF (NUMLIT[J] LEQ NUMLIT[J+1]) THEN
                  BEGIN INPT[J] SWAP INPT[J+1];
                        NUMLIT[J] SWAP NUMLIT[J+1];
                  END;
               J_J+1;
            END;
   END;
  END "SORT SUBROUTINE";

  INTEGER NFUN,NBIN,NUNION,NPTR2,COUNT,K,COUNT2;
  STRING ARRAY PHI[1:250],INTEM[1:250],UTEM[1:250], ISTB[1:250];
  STRING BUF1,BUF2,BTERM,BTEM,NEWGAT,TEMPX,BUFFER,BUF3;
  BOOLEAN COVER;
  COMMENT ***MEANINGS OF VARIBLES***
    PHI = ARRAY OF INPUT LIST
    INTEM = ARRAY OF PRODUCT TERMS REPRESENTING INTERSECTIONS
    UTEM = ARRAY OF INTERSECTIONS AFTER FACTORING OUT THE P.T.
    NFUN = # OF NON-EMPTY P.T. THAT REPRESENT INTERSECTIONS
    NUNION = # OF TERMS TO UNION TOGETHER
    NBIN = # OF NON-EMPTY INTERSECTIONS WITH LISTB
    COUNT = #  OF GATES IN LIST B ;
   INTEGER ARRAY PTR2[1:14],PTR3[1:14],PTR4[1:8];
   INTEGER CCOUNT,NUMB,NANS,MMS;
   STRING ARRAY ANS[1:250];
   INTEGER STAGE,NN,N2,N3,N4,UPBOUND;
   INTEGER ARRAY GATCON[1:250];
   BOOLEAN ALDONE,BACKTK,ANSGOOD,REONFG;
   STRING ARRAY STACK[1:130,1:130];
   LABEL TRYAG,FINANS;


    FORWARD PROCEDURE TAUTOLOGY;
    FORWARD PROCEDURE INTERB;
    FORWARD PROCEDURE INTERF;
    FORWARD PROCEDURE UNION;
    FORWARD PROCEDURE CHOX;
    FORWARD PROCEDURE PAREN;
    FORWARD PROCEDURE UPTAB2;



    PROCEDURE INLIST;
    COMMENT ******* TO DETERMINE IF THE P.T. IS COVERED *****;
    BEGIN "INLIST";
    K_1;
    COUNT_1;
    LISTB[1]_PHI[1];
    CHOX;
    BTERM_LISTB[1];
    PAREN;
    LISTB[1]_BTERM;
    UPTAB2;
    FOR K_2 STEP 1 UNTIL NPT DO
    BEGIN TEMP_PHI[K];
          INTERB;
          UNION;
         IF (NOT COVER)THEN BEGIN COUNT_COUNT+1;
                                  LISTB[COUNT]_NEWGAT;
                           END;
```

```
                CHOX;
                BTERM_LISTB[COUNT];
                PAREN;
                LISTB[COUNT]_BTERM;
                UPTAB2;
END;
FOR K_1 STEP 1 UNTIL COUNT DO
  BEGIN NUMLIT[K]_0;
        BUF_LISTB[K];
        FOR J_NF+1 STEP 1 UNTIL L DO
          IF (BUF[J FOR 1] = "0") OR (BUF[J FOR 1] = "1")
          THEN NUMLIT[K]_NUMLIT[K]+1;
  END;
FOR I_1 STEP 1 UNTIL COUNT-1 DO
  BEGIN J_1;
        WHILE (J LEQ COUNT-1) DO
          BEGIN IF (NUMLIT[J] LEQ NUMLIT[J+1]) THEN
              BEGIN LISTB[J] SWAP LISTB[J+1];
                    NUMLIT[J] SWAP NUMLIT[J+1];
              END;
            J_J+1;
          END;
  END;
END "INLIST";


PROCEDURE CHOX;
COMMENT ******* TO DETERMINE IF X OR X' SHOULD BE TAKEN *****;
BEGIN "CHOX"
INTEGER M1;
BUF1_LISTB[COUNT];
BUF2_PHI[K];
PHI[K]_"0";
FOR M1_2 STEP 1 UNTIL L DO
  PHI[K]_PHI[K]&"2";
FOR M1_NF+1 STEP 1 UNTIL L DO
 BEGIN  IF (BUF1[M1 FOR 1] = "*") THEN
         BEGIN BUF1_BUF1[1 FOR M1-1]&"2"&BUF1[M1+1 FOR L-M1];
               TEMP_BUF1;
               INTERF;
               UNION;
               IF (NOT COVER)THEN
                  LISTB[COUNT]_LISTB[COUNT][1 FOR M1-1]&"2"
                                  &LISTB[COUNT][M1+1 FOR L-M1]
               ELSE LISTB[COUNT]_LISTB[COUNT][1 FOR M1-1]&"3"
                                    &LISTB[COUNT][M1+1 FOR L-M1];
         END;
 END;
PHI[K]_BUF2;
END "CHOX";


PROCEDURE PAREN;
COMMENT ****** TO DETERMINE WHICH VARIABLE CAN BE PARENTHESIZED **** ;
BEGIN "PAREN"
INTEGER J;
BTEM_BTERM;
FOR J_NF+1 STEP 1 UNTIL L DO
 IF (BTERM[J FOR 1] = "2") THEN
      BTERM_BTERM[1 FOR J-1]&"0"&BTERM[J+1 FOR L-J]
```

```
ELSE IF (BTERM[J FOR 1] = "3") THEN
     BTERM_BTERM[1 FOR J-1]&"1"&BTERM[J+1 FOR L-J];
FOR J_NF+1 STEP 1 UNTIL L DO
  IF (BTEM[J FOR 1] NEQ "2") AND (BTEM[J FOR 1] NEQ "3") THEN
     BEGIN  TEMP_BTERM;
            IF (TEMP[J FOR 1] = "0")THEN
            TEMP_TEMP[1 FOR J-1]&"2"&TEMP[J+1 FOR L-J]
            ELSE IF(TEMP[J FOR 1] = "1")THEN
            TEMP_TEMP[1 FOR J-1]&"3"&TEMP[J+1 FOR L-J];
            BUF_TEMP;
            INTERF;
            UNION;
            IF COVER THEN  BTEM_BTEM[1 FOR J-1]&BUF[J FOR 1]
                                &BTEM[J+1 FOR L-J];
     END ;
BTERM_BTEM;
END "PAREN";



PROCEDURE INTERF;
COMMENT **** FIND THE INTERSECTIONS OF A P.T. AND THE FUNCTIONS ***;
BEGIN "INTERF"
INTEGER I,I1,I2;
BOOLEAN FLAGX;
TEMPX_TEMP;
NFUN_0;
FOR I_1 STEP 1 UNTIL NPT DO
BEGIN TEMP_TEMPX;
      FOR I1_1 STEP 1 UNTIL NF DO
      IF (TEMP[I1 FOR 1] = "1")
      AND (PH1[I][I1 FOR 1] = "1")THEN
       BEGIN  FLAGX_TRUE;
            FOR I2_NF+1 STEP 1 UNTIL L DO
            BEGIN  IF (PH1[I][I2 FOR 1] = "-")
                   OR (PH1[I][I2 FOR 1] = TEMP[I2 FOR 1])
                   THEN CONTINUE
                   ELSE IF (TEMP[I2 FOR 1] = "-")
                        OR (TEMP[I2 FOR 1] = "2")
                        OR (TEMP[I2 FOR 1] = "3")
                        THEN TEMP_TEMP[1 FOR I2-1]&PH1[I][I2 FOR 1]
                                   &TEMP[I2+1 FOR L-I2]
                        ELSE BEGIN FLAGX_FALSE;
                                   I2_L+1;
                             END;
            END ;
            IF FLAGX THEN BEGIN NFUN_NFUN+1;
                             INTEM[NFUN]_TEMP;
                       END;
       END ;
END ;
NUNION_NFUN;
END "INTERF";


PROCEDURE INTERB;
COMMENT *** FIND THE INTERSECTION OF P.T. AND LIST B ***;
BEGIN "INTERB"
INTEGER J,J1,J2;
BOOLEAN FLAGX;
```

```
TEMPX←TEMP;
NBIN←0;
FOR J←1 STEP 1 UNTIL COUNT DO
  BEGIN TEMP←TEMPX;
          FOR J1←1 STEP 1 UNTIL NF DO
          IF (TEMP[J1 FOR 1] = "1") AND (LISTB[J][J1 FOR 1] NEQ "0") THEN
          BEGIN  FLAGX←TRUE;
              FOR J2←NF+1 STEP 1 UNTIL L DO
               BEGIN  IF (LISTB[J][J2 FOR 1] = TEMP[J2 FOR 1])
                          OR (LISTB[J][J2 FOR 1] = "-")
                          OR (LISTB[J][J2 FOR 1] = "2")
                          OR (LISTB[J][J2 FOR 1] = "3")THEN CONTINUE
                          ELSE IF (TEMP[J2 FOR 1] = "-")
                               THEN TEMP←TEMP[1 FOR J2-1]&LISTB[J][J2 FOR 1]
                                          &TEMP[J2+1 FOR L-J2]
                               ELSE BEGIN FLAGX←FALSE;
                                          J2←L+1;
                                    END;
               END ;
              IF FLAGX THEN  BEGIN NBIN←NBIN+1;
                                   INTEM[NBIN]←TEMP;
                             END;
        END ;
  END ;
NUNION←NBIN;
END "INTERB";



PROCEDURE UNION;
COMMENT *** FIND A LIST OF PRODUCT TERMS TO BE UNION TOGETHER ***;
BEGIN "UNION"
LABEL LOOP;
INTEGER M1,M2,JX;
IF (NUNION = 0)THEN  BEGIN COVER←FALSE;
                           NEWGAT←TEMPX;
                           GO TO LOOP;;
                     END;
NPTR2←0;
FOR M1←NF+1 STEP 1 UNTIL L DO
  BEGIN IF (TEMPX[M1 FOR 1] NEQ "0" AND TEMPX[M1 FOR 1] NEQ "1")THEN
          BEGIN NPTR2←NPTR2+1;
                PTR2[NPTR2]←M1;
          END;
  END;
IF (NPTR2 NEQ 0)THEN
BEGIN FOR M2←1 STEP 1 UNTIL NUNION DO
          FOR M1←1 STEP 1 UNTIL NPTR2 DO
            BEGIN JX←PTR2[M1];
                  UTEM[M2]←UTEM[M2]&INTEM[M2][JX FOR 1];
            END;
      TAUTOLOGY;
END
ELSE COVER←TRUE;
LOOP: FOR M2←1 STEP 1 UNTIL NUNION DO
          UTEM[M2]←"";
END "UNION";
```

```
PROCEDURE TAUTOLOGY;
COMMENT *** DECIDE IF A LIST OF PRODUCT TERMS HAVE A SUM OF LOGICAL 1 ***;
BEGIN "TAUTOLOGY"
INTEGER LASTP,I,J1,M;
STRING TESTC;
BOOLEAN F1;
TESTC_"0";
F1_TRUE;
FOR J1_2 STEP 1 UNTIL NPTR2 DO
   TESTC_TESTC&"-";
I_1;
LASTP_1;
BEGIN LABEL LOOP2,LOOP3;
LOOP2: J1_1;
 LOOP3: IF (UTEM[I][J1 FOR 1] = "-")
        OR (UTEM[I][J1 FOR 1] = "2")
        OR (UTEM[I][J1 FOR 1] = "3")
        OR (TESTC[J1 FOR 1] = UTEM[I][J1 FOR 1]) THEN
           BEGIN COVER_TRUE;
                 J1_J1+1;
                 IF (J1 LEQ NPTR2) THEN GO TO LOOP3;
           END
        ELSE BEGIN COVER_FALSE;
                   I_I+1;
                   IF ( I LEQ NUNION) THEN GO TO LOOP2
                   ELSE IF (LASTP NEQ NPTR2) THEN
                        BEGIN LASTP_LASTP+1;
                              TESTC_TESTC[1 FOR LASTP-1]&"0"
                                   &TESTC[LASTP+1 FOR NPTR2-LASTP];
                              I_1;
                              GO TO LOOP2;
                        END;
             END;
        IF COVER THEN
        BEGIN  IF (TESTC[LASTP FOR 1] NEQ "1") THEN
             BEGIN TESTC_TESTC[1 FOR LASTP-1]&"1"&TESTC[LASTP+1 FOR NPTR2-LASTP];
                   I_1;
                   GO TO LOOP2;
             END
           ELSE WHILE F1 DO BEGIN IF (LASTP = 1) THEN F1_FALSE;
                                  TESTC_TESTC[1 FOR LASTP-1]&"-"&TESTC[LASTP+1 FOR NPTR2-LASTP];
                                  LASTP_LASTP-1;
                                  IF (TESTC[LASTP FOR 1] = "0") THEN
                                  BEGIN TESTC_TESTC[1 FOR LASTP-1]&"1"&TESTC[LASTP+1 FOR NPTR2-LASTP];
                                        I_1;
                                        GO TO LOOP2;
                                  END;
                            END;
        END
        ELSE BEGIN FOR I_1 STEP 1 UNTIL NPTR2 DO
                   BEGIN M_PTR2[I];
                         TEMPX_TEMPX[1 FOR M-1]&TESTC[I FOR 1]
                              &TEMPX[M+1 FOR L-M];
                   END;
                   NEWGAT_TEMPX;
             END;
END;
END "TAUTOLOGY";
```

```
PROCEDURE UPTAB2;
COMMENT *** UPDATE THE TABLE OF USEFULNESS ***;
BEGIN "UPTAB"
INTEGER K2,K3,K4,K5;
BOOLEAN FX1,FX2;
   BUF1_LISTB[COUNT];
      FOR K2_NF+1 STEP 1 UNTIL L DO
        IF (BUF1[K2 FOR 1] = "2") THEN
             BUF1_BUF1[1 FOR K2-1]&"0"&BUF1[K2+1 FOR L-K2]
        ELSE IF (BUF1[K2 FOR 1] = "3") THEN
             BUF1_BUF1[1 FOR K2-1]&"1"&BUF1[K2+1 FOR L-K2];
     FOR K3_1 STEP 1 UNTIL NF DO
      BEGIN  IF (BUF1[K3 FOR 1] NEQ "1") THEN
          BEGIN FX1_FALSE;
                FOR K2_NF+1 STEP 1 UNTIL L DO
                  IF ( BUF1[K2 FOR 1] = "-")THEN
                   BEGIN FX1_TRUE;
                          K2_L+1;
                   END;
 COMMENT *** IF THERE ARE '-' IN THE TERM, USE INTERF ***
              ZERO ALL THE OTHER OUTPUT FUNCTIONS EXCEPT THE ONE
              THAT IS TO BE TESTED ;

         IF (FX1) THEN
              BEGIN TEMP_BUF1;
                    FOR K2_1 STEP 1 UNTIL NF DO
                      IF (K2 NEQ K3) THEN TEMP_TEMP[1 FOR K2-1]&"0"
                                          &TEMP[K2+1 FOR L-K2]
                                  ELSE TEMP_TEMP[1 FOR K2-1]&"1"
                                          &TEMP[K2+1 FOR L-K2];

                    INTERF;
                    UNION;
                    IF (COVER) THEN LISTB[COUNT]_LISTB[COUNT][1 FOR K3-1]&"7"
                                          &LISTB[COUNT][K3+1 FOR L-K3];

              END

         ELSE BEGIN FX2_FALSE;
                FOR K4_1 STEP 1 UNTIL NPT DO
                 IF(PHI[K4][K3 FOR 1] = "1") THEN
                 BEGIN FX2_TRUE;
                    FOR K5_NF+1 STEP 1 UNTIL L DO
                     IF(BUF1[K5 FOR 1] NEQ PHI[K4][K5 FOR 1])
                      AND (PHI[K4][K5 FOR 1] NEQ "-")THEN BEGIN FX2_FALSE;
                                                          K5_L+1;
                                                       END;

                   IF FX2 THEN  BEGIN K4_NPT+1;
                                   LISTB[COUNT]_LISTB[COUNT][1 FOR K3-1]&"7"
                                          &LISTB[COUNT][K3+1 FOR L-K3];
                                END;
                 END;
              END;
          END;
      END ;
 END "UPTAB";



 PROCEDURE REDUN;
 COMMENT *** DELETE ALL THE GATES THAT ARE REDUNDANT *** ;
```

```
BEGIN "REDUN"
INTEGER I,J,K;
REDNFG_FALSE;
FOR I_COUNT STEP -1 UNTIL 1 DO
 BEGIN BUFFER_LISTB(I);
       LISTB(I)_"0";
       FOR K_1 STEP 1 UNTIL NPT DO
       FOR J_1 STEP 1 UNTIL NF DO
       IF (BUFFER(J FOR 1) NEQ "0") AND (PHI(K)(J FOR 1) = "1") THEN
        BEGIN TEMP_PHI(K);
              INTERB;
              UNION;
              IF (NOT COVER) THEN BEGIN K_NPT+1;
                                        LISTB(I)_BUFFER;
                                  END;
              J_NF+1;
        END;
       IF (COVER) THEN BEGIN FOR J_I STEP 1 UNTIL COUNT-1 DO
                                LISTB(J)_LISTB(J+1);
                             COUNT_COUNT-1;
                             REDNFG_TRUE;
                       END;
 END;
END "REDUN";
COMMENT ******* PROGRAM STATEMENTS FOR PHASE1 BEGIN ******* ;
COMMENT  SET I/O ;
LINE_1 ;
SETBREAK(LINE,'15,'12,"INS");
COUNT1_13; EOF_0;
CHAN1_GETCHAN;
OPEN(CHAN1,"DSK",0,2,0,COUNT1,BRCHAR,EOF);
LOOKUP(CHAN1,"DATA0",FLAG);
CHAN2_GETCHAN;
OPEN(CHAN2,"DSK",0,2,2,0,0,0);
ENTER (CHAN2,"OUT2",FLAG);
COMMENT *** END I/O *** ;
BUF_INPUT(CHAN1,LINE);
NV_INTSCAN(BUF,BRCHAR);
BUF_INPUT(CHAN1,LINE);
NF_INTSCAN(BUF,BRCHAR);
L_NV+NF;
PTRP_NF+1; PTRF_1;
INPT(1)_INPUT(CHAN1,LINE);
NPT_1; FLAG1_FALSE;
NPTX_1;
PHI(1)_INPT(1);
OUT(CHAN2,"THE INPUT PRODUCT TERMS ARE:"&'15&'12);
OUT (CHAN2,INPT(1)&'15&'12);
WHILE (NOT FLAG1) DO
  BEGIN   BUF_INPUT(CHAN1,LINE);
          OUT(CHAN2,BUF&'15&'12);
          IF (BUF NEQ "999") THEN
            BEGIN NPT_NPT+1;
                  PHI(NPT)_BUF;
                  FOR I_1 STEP 1 UNTIL NPTX DO
                   BEGIN COMPARE(BUF,INPT(I));
                      IF FLAG2 THEN
```

```
                            BEGIN UPTAB(INPT[I],BUF);
                                  INPT[I]_INPT[I][1 FOR J-1]&"1"&INPT[I][J+1 FOR L-J];
                                  I_NPTX+1;
                            END;
                      END;
                    IF NOT FLAG2 THEN BEGIN NPTX_NPTX+1;
                                            INPT[NPTX]_BUF;
                                      END;
              END
             ELSE FLAG1_TRUE;
  END;
COMMENT END OF INPUT, BEGIN SORTING *** ;
 SORT;
COMMENT OUTPUT THE SORTED PRODUCT TERMS;
BUF_"THE SORTED PRODUCT TERMS ARE:";
OUT(CHAN2,BUF&'15&'12);
FOR I_1 STEP 1 UNTIL NPTX DO
  OUT(CHAN2,INPT[I]&'15&'12);
BUF_CVS(NPTX);
OUT (CHAN2,"THE UPPER BOUND IS: "&BUF&'15&'12);
OUTSTR ("THE UPPER BOUND IS: "&BUF&'15&'12);
CLOSE(CHAN1);



COMMENT ***** STATEMENTS OF PHASE1 ****;
INLIST;



COMMENT *** IF A TERM IS USEFUL FOR ALL THE FUNCIIONS IT IS POTENTIALLY USEFUL THEN TRANSFORM
            THE TERM TO ITS MAXIMUM CUBE ***;
FOR I_1 STEP 1 UNTIL COUNT DO
BEGIN NUMB_0;
       FOR J_1 STEP 1 UNTIL NF DO
            IF (LISTB[I][J FOR 1] NEQ "0") THEN NUMB_NUMB+1;
      IF (NUMB = 1) THEN
      BEGIN BUFFER_LISTB[I];
             FOR K_1 STEP 1 UNTIL L DO
                 IF (BUFFER[K FOR 1] = "2") OR (BUFFER[K FOR 1] = "3")THEN
                 BUFFER_BUFFER[1 FOR K-1]&"-"&BUFFER[K+1 FOR L-K];
            TEMP_BUFFER;
            INTERF;
            UNION;
           IF (COVER) THEN LISTB[I]_BUFFER;
      END
      ELSE BEGIN BUF3_LISTB[I];
                 BUFFER_LISTB[I];
                 FOR K_1 STEP 1 UNTIL L DO
                   IF (BUFFER[K FOR 1] = "2") OR (BUFFER[K FOR 1] = "3")
                   THEN BUFFER_BUFFER[1 FOR K-1]&"-"&BUFFER[K+1 FOR L-K];
                 FOR J_1 STEP 1 UNTIL NF DO
                   BUFFER_BUFFER[1 FOR J-1]&"0"&BUFFER[J+1 FOR L-J];
                 N3_ COUNT;
                 COUNT_I;
                 LISTB[COUNT]_BUFFER;
                 UPTAB2;
                 BUFFER_LISTB[COUNT];
                 COUNT_N3;
                 FLAG1_FALSE;
```

```
                    FOR J_1 STEP 1 UNTIL NF DO
                      IF(BUF3[J FOR 1] NEQ "0") AND (BUFFER[J FOR 1]="0")
                      THEN BEGIN FLAG1_TRUE;
                                 J_NF+1;
                           END;
                    IF (FLAG1) THEN LISTB[I]_BUF3
                               ELSE LISTB[I]_BUFFER;
       END;
END;

BUF1_"THE LIST OBTAINED AT THE END OF PHASE1";
OUTSTR ("THE LIST OBTAINED AT THE END OF PHASE 1 IS:"&'15&'12);
OUT(CHAN2,BUF1&'15&'12);
FOR K_1 STEP 1 UNTIL COUNT DO
  BEGIN OUTSTR (LISTB[K]&'15&'12);
        OUT(CHAN2,LISTB[K]&'15&'12);
  END;
LBOUND_COUNT;
BUF1_CVS(LBOUND);
OUT (CHAN2,"THE LOWER BOUND IS : "&BUF1&'15&'12);
OUTSTR ("THE LOWER BOUND IS : "&BUF1&'15&'12);

COMMENT *** END OF PHASE 1 ***;



COMMENT *** BEGIN BRANCH-AND-BOUND SEARCH OF OPTIMAL SOLUTION ***;
GATCON[1]_COUNT;
STAGE_1;
FOR I_1 STEP 1 UNTIL COUNT DO
  STACK[1,I]_LISTB[I];
ALDONE_FALSE;
BACKTK_FALSE;
UPBOUND_NPTX;

TRYAG:  WHILE (NOT ALDONE) DO
  BEGIN "BRANCHING"
      IF (NOT BACKTK) THEN
       BEGIN "NOT BACK-TRACK"
         FOR I_1 STEP 1 UNTIL GATCON[STAGE] DO
          BEGIN FOR J_NF+1 STEP 1 UNTIL L DO
              IF (STACK[STAGE,I][J FOR 1] = "2")
               OR (STACK[STAGE,I][J FOR 1] = "3")
              THEN BEGIN
                     STACK[STAGE+1,I]_STACK[STAGE,I][1 FOR J-1]&"-"
                                     &STACK[STAGE,I][J+1 FOR L-J];
                     IF (STACK[STAGE,I][J FOR 1] = "2")
                     THEN STACK[STAGE,I]_STACK[STAGE,I][1 FOR J-1]&"X"
                                    &STACK[STAGE,I][J+1 FOR L-J]
                      ELSE STACK[STAGE,I]_STACK[STAGE,I][1 FOR J-1]&"Y"
                                         &STACK[STAGE,I][J+1 FOR L-J];

                     FOR K_1 STEP 1 UNTIL GATCON[STAGE] DO
                         IF (K NEQ I) THEN STACK[STAGE+1,K]_STACK[STAGE,K];

                     NN_I;
                     I_GATCON[STAGE]+1;
                     J_L+1;
                     ALDONE_FALSE;
                   END
```

```
            ELSE ALDONE_TRUE;
          END;
        IF (ALDONE) THEN GO TO FINANS;
        GATCON[STAGE+1]_GATCON[STAGE];
        BUFFER_CVS(STAGE);
        OUT (CHAN2,"STACK "&BUFFER&" IS:"&'15&'12);
        OUTSTR ("STACK "&BUFFER&" IS:"&'15&'12);
        J_0;
        MMS_GATCON[STAGE];
        WHILE (J LEQ MMS) DO
          BEGIN FOR K_1 STEP 1 UNTIL 10 DO
                BEGIN J_J+1;
                      IF (J LEQ MMS) THEN OUT (CHAN2,STACK[STAGE,J]&" ")
                                     ELSE K_11;
                END;
              OUT (CHAN2,'15&'12);
          END;
        STAGE_STAGE+1;
        COUNT_GATCON[STAGE];
        FOR I_1 STEP 1 UNTIL COUNT DO
          LISTB[I]_STACK[STAGE,I];

    COMMENT *** CHECK FOR THE CORRECTNESS OF THE []-VARIABLES OF THE
                TRANSFORMED TERM *** ;

        FOR I_NF+1 STEP 1 UNTIL L DO
          BEGIN TEMP_LISTB[NN];
                IF (TEMP[I FOR 1] = "2") OR (TEMP[I FOR 1] = "3")
                THEN BEGIN
                        TEMP_TEMP[1 FOR I-1]&"-"&TEMP[I+1 FOR L-I];
                        FOR J_NF+1 STEP 1 UNTIL L DO
                          IF (J NEQ I) AND (TEMP[J FOR 1] = "2")
                          THEN TEMP_TEMP[1 FOR J-1]&"0"&TEMP[J+1 FOR L-J]
                          ELSE IF (J NEQ I) AND (TEMP[J FOR 1] = "3")
                          THEN TEMP_TEMP[1 FOR J-1]&"1"&TEMP[J+1 FOR L-J];

                        INTERF;
                        UNION;
                        IF (NOT COVER) THEN
                        BEGIN IF (LISTB[NN][I FOR 1]="2")
                              THEN LISTB[NN]_LISTB[NN][1 FOR I-1]&"0"&LISTB[NN][I+1 FOR L-I]
                              ELSE IF (LISTB[NN][I FOR 1] = "3")
                                   THEN LISTB[NN]_LISTB[NN][1 FOR I-1]&"1"&LISTB[NN][I+1 FOR L-I];
                        END;
                     END;
          END;


    COMMENT *** UPDATE THE TABLE OF USEFULNESS FOR THE TRANSFORMED TERM *** ;
          BUF3_LISTB[NN];
          FOR I_1 STEP 1 UNTIL NF DO
            IF (LISTB[NN][I FOR 1] NEQ "1")
            THEN LISTB[NN]_LISTB[NN][1 FOR I-1]&"0"&LISTB[NN][I+1 FOR L-I];
          N3_COUNT;
          COUNT_NN;
          UPTAB2;
          COUNT_N3;

    COMMENT *** IF THE TRANSFORMED TERM IS USEFUL FOR ONLY ONE OUTPUT FUNCTION
                THEN LET IT COVER AS MANY MINTERMS AS POSSIBLE *** ;
```

```
        NUMB_0;
        FOR J_1 STEP 1 UNTIL NF DO
          IF (LISTB[NN][J FOR 1] NEQ "0") THEN NUMB_NUMB+1;
        IF (NUMB = 1) THEN
         BEGIN BUFFER_LISTB[NN];
               FOR K_NF+1 STEP 1 UNTIL L DO
                  IF (BUFFER[K FOR 1] = "2") OR (BUFFER[K FOR 1] = "3")
                  THEN BUFFER_BUFFER[1 FOR K-1]&"-"&BUFFER[K+1 FOR L-K];
              TEMP_BUFFER;
              INTERF;
              UNION;
              IF (COVER) THEN LISTB[NN]_BUFFER;
         END;
      END "NOT BACK-TRACK"


COMMENT *** BACK-TRACKING *** ;
ELSE BEGIN STAGE_STAGE-1;
            OUT (CHAN2,"BACK-TRACKING"&'15&'12);
            OUTSTR ("BACK-TRACKING"&'15&'12);
            IF (STAGE LEQ 0) OR (LBOUND = UPBOUND) THEN BEGIN ANSGOOD_TRUE;
                                         GO TO FINANS;
                                                   END;
               WHILE (GATCON[STAGE] GEQ UPBOUND) DO
            BEGIN OUT (CHAN2,"BACK-TRACKING: GATCON[STAGE] GEQ UPBOUND"&'15&'12);
               OUTSTR ("BACK-TRACKING: GATCON[STAGE] GEQ UPBOUND"&'15&'12);
               IF (STAGE LEQ 1) THEN BEGIN ANSGOOD_TRUE;
                                           GO TO FINANS;
                                     END;
               STAGE_STAGE-1;
            END;
               COUNT_GATCON[STAGE];
               FOR I_1 STEP 1 UNTIL COUNT DO
                  BEGIN FOR J_NF+1 STEP 1 UNTIL L DO
                      IF (STACK[STAGE,I][J FOR 1] = "X")
                       OR (STACK[STAGE,I][J FOR 1] = "Y")
                      THEN BEGIN IF (STACK[STAGE,I][J FOR 1] = "X")
                                 THEN STACK[STAGE,I]_STACK[STAGE,I][1 FOR J-1]&"0"
                                                     &STACK[STAGE,I][J+1 FOR L-J]
                                 ELSE STACK[STAGE,I]_STACK[STAGE,I][1 FOR J-1]&"1"
                                                     &STACK[STAGE,I][J+1 FOR L-J];

                                 NN_I;
                                 BUF3_LISTB[NN];
                                 J_L+1;
                                 I_COUNT+1;
                           END;
                  END;
               FOR I_1 STEP 1 UNTIL COUNT DO
                  LISTB[I]_STACK[STAGE,I];
         END;



COMMENT *** FIND NEW GATES AND CHECK REDUNDANCY ***;
COMMENT    *** CHECK FOR BOTH FORWARD BRANCHING AND BACK-TRACKING *** ;
FOR K_1 STEP 1 UNTIL NPT DO
 FOR I_1 STEP 1 UNTIL NF DO
  IF (BUF3[I FOR 1] NEQ "0") AND (PH1[K][I FOR 1] = "1") THEN
  BEGIN TEMP_PH1[K];
```

```
            INTERB;
            UNION;
            IF (NOT COVER) THEN BEGIN COUNT_COUNT+1;
                                      LISTB[COUNT]_NEWGAT;
                                      CHOX;
                                      BTERM_LISTB[COUNT];
                                      PAREN;
                                      LISTB[COUNT]_BTERM;
                                      UPTAB2;
                                 END;
            I_NF+1;
     END;
   IF (COUNT GEQ UPBOUND)  THEN BACKTK_TRUE
   ELSE BEGIN BACKTK_FALSE;
              GATCON[STAGE]_COUNT;
              FOR I_1 STEP 1 UNTIL COUNT DO
                STACK[STAGE,I]_LISTB[I];
        END;
   END "BRANCHING";
   FIMANS: IF (ANSGOOD) THEN BEGIN OUT (CHAN2,"THE OPTIMAL REALIZATION IS:"&'15&'12);
                                        FOR I_1 STEP 1 UNTIL UPBOUND DO
                                          OUT(CHAN2,ANS[I]&'15&'12);
                                END
             ELSE BEGIN REDUN;
                        IF (COUNT LEQ UPBOUND)THEN
                    BEGIN UPBOUND_COUNT;
                        BUFFER_CVS(UPBOUND);
                        OUT (CHAN2,"THE UPPER BOUND IS NOW: "&BUFFER&'15&'12);
                        OUTSTR ("THE UPPER BOUND IS NOW: "&BUFFER&'15&'12);
                        OUT (CHAN2,"AN IMPROVED SOLUTION IS:"&'15&'12);
                        OUTSTR ("AN IMPROVED SOLUTION IS:"&'15&'12);
                        FOR I_1 STEP 1 UNTIL COUNT DO
                          BEGIN OUT (CHAN2,LISTB[I]&'15&'12);
                                OUTSTR (LISTB[I]&'15&'12);
                                ANS[I]_LISTB[I];
                          END;
                    END;
                        BACKTK_TRUE;
                        ALDONE_FALSE;
                        GO TO TRYAG;
                  END;
   CLOSE (CHAN2);
   END "MINI";
```

Appendix B

Test problem specifications

and solutions

```
EXAMPLE 1 : 7 SEGMENT DECODER
            4 INPUT VARIABLES
            7 OUTPUT VARIABLES


THE SORTED PRODUCT TERMS ARE :
00000100100
00001000101
00000100111
001000001-1
0011100001-
10000000-00
1011000100-
1001000010-
100000001-0
0110100-000
01011000-10
00000010--1
000000101--
000001000--
0000011-00-
THE UPPER BOUND IS: 15


THE OPTIMAL REALIZATION IS:
0071770001-
100700701-0
7077077100-
1770777-000
01077000-10
00100770-11
70771070101
70000170-00
0000017-00-
```

```
EXAMPLE 2 : FAST SHIFTER
                13 input variables
                8 output variables


THE SORTED PRODUCT TERMS ARE:
001000001101110000000
000001011111110000000
00000010-101000001000
00000010-100100000100
00000001-111001000000
00000010-000100000001
00000100-110110000000
00000100-110001000000
00000100-101100100000
00000100-101000010000
00000100-100100001000
00000001-110100100000
00000100-000100000010
00000100-001000000001
000011001111-10000000
00001000-110010000000
00001000-101101000000
00001000-101000100000
00001000-100100010000
00000001-110000010000
00001000-000100000100
00001000-001000000010
00001000-001100000001
00001000111-110000000
00010000-101110000000
00010000-101001000000
00010000-100100100000
00000001-101100001000
00010000-000100001000
00010000-001000000100
00010000-001100000010
00010000-010000000001
00000001-101000000100
00100000-101010000000
00100000-100101000000
00000001-100100000010
00100000-000100010000
00100000-001000001000
00100000-001100000100
00100000-010000000010
00100000-010100000001
01000000-100110000000
00000001-111110000000
01000000-000100100000
01000000-001000010000
01000000-001100001000
01000000-010000000100
01000000-010100000010
```

```
01000000-011000000001
00000010-111010000000
00000010-110101000000
10000000-000101000000
10000000-001000100000
10000000-001100010000
10000000-010000001000
10000000-010100000100
10000000-011000000010
10000000-011100000001
00000010-110000100000
00000010-101100010000
00000001--00000000001
00000010--00000000010
00000100--00000000100
00001000--00000001000
00010000--00000010000
01110000111--10000000
00100000--00000100000
01000000--00001000000
0100000011-1-10000000
10000000--00010000000
100000001----10000000
THE UPPER BOUND IS: 71
THE OPTIMAL REALIZATION IS:
00000010-110000100000
00000010-101100010000
00000010-101000001000
00000010-100100000100
00000001-111001000000
00000010-000100000001
777777101111-10000000
00000100-110110000000
00000100-110001000000
00000100-101100100000
00000100-101000010000
00000100-100100001000
00000001-110100100000
00000100-000100000010
00000100-001000000001
00001000-110010000000
00001000-101101000000
00001000-101000100000
00001000-100100010000
00000001-110000010000
00001000-000100000100
```

```
10200   00001000-001000000010
10300   00001000-001100000001
10400   00000001-101100001000
10500   00010000-101110000000
10600   00010000-101001000000
10700   00010000-100100100000
10800   00000001-101000000100
10900   00010000-000100001000
11000   00010000-001000000100
11100   00010000-001100000010
11200   00010000-010000000001
11300   00100000-101010000000
11400   00100000-100101000000
11500   00000001-100100000010
11600   00100000-000100010000
11700   00100000-001000001000
11800   00100000-001100000100
11900   00100000-010000000010
12000   00100000-010100000001
12100   00000001-111110000000
12200   01000000-100110000000
12300   00000010-111010000000
12400   01000000-000100100000
12500   01000000-001000010000
12600   01000000-001100001000
12700   01000000-010000000100
12800   01000000-010100000010
12900   01000000-011000000001
13000   00000010-110101000000
13100   10000000-000101000000
13200   10000000-001000100000
13300   10000000-001100010000
13400   10000000-010000001000
13500   10000000-010100000100
13600   10000000-011000000010
13700   10000000-011100000001
13800   771000000011-1-10000000
13900   01000000--00001000000
14000   10000000--00010000000
14100   00000001--00000000001
14200   00000010--00000000010
14300   00000100--00000000100
14400   00001000--00000001000
14500   77771000111--10000000
14600   00010000--00000010000
14700   00100000--00000100000
14800   100000001----10000000
14900   TOTAL OF 68 gates
```

```
EXAMPLE 3 : 2 DIGIT BCD TO BINARY DECODER
            8 INPUT VARIABLES
            7 OUTPUT FUNCTIONS


THE SORTED PRODUCT TERMS ARE:
00001100111000-
00000100111010-
00011100111100-
00000101000001-
00000101000011-
00000101001000-
00111101001010-
01000101001100-
00000100000001-
00001000001001-
00000100000011-
00011000010100-
00001000011011-
00000100001000-
00011000101001-
00011100001010-
10001000110100-
00001000111011-
00100100001100-
00001001001001-
00010000000100-
00000100010001-
00000100010011-
00111100011000-
00010000111001-
00000100011010-
00010001000100-
01001100011100-
00100000001011-
00000100100001-
00100000100100-
00000100100011-
00000100101000-
01000000011001-
00011100101010-
00000100101100-
01000001001011-
00000100110001-
00000100110011-
```

```
0001000011101--
0011000100100--
0000100000001--
0000100001000--
0001100010001--
0100000001101--
0111100011000--
1000000011001--
0000100100001--
0001000000100--
0001000001001--
0001000010000--
10100001000----
01000000100----
10000000111----
10000001001----
00100000010----
01100000101----
0000001-------1
THE UPPER BOUND IS: 58
THE UPPER BOUND IS NOW : 40
AN IMPROVED SOLUTION IS:
77000701001100-
07007700011100-
77000001001011-
01000000110-1-
0100000001101--
00700700-01100-
00001000-11011-
7000700011-100-
0770000010-100-
0077000001001--
00100000-01011-
1000000011-01--
0007100-001001-
0001000-000100-
000100001--001-
10000000111----
00100000010----
07100000101----
070100001-000--
7010000100-0-0-
0100000010-----
0000010-0000-1-
0000100-00-010-
00001000-1--00-
0000001-------1
00071000-01001-
000100000-0100-
```

```
09700    7001000011101--
09800    0701700010-010-
09900    70100001000----
10000    1000000100-----
10100    7010000100-00--
10200    00000100--00-1-
10300    0000100-00001--
10400    00177700011000-
10500    0010700-1000--
10600    00000100--10-0-
10700    70077100111100-
10800    0007010-0010-0-
10900    0701700010001--
```

```
EXAMPLE 4 : 3 BIT BINARY MULTIPLIER
            6 INPUT VARIABLES
            6 OUTPUT FUNCTIONS


THE SORTED PRODUCT TERMS ARE :
110001111111
000001001001
000010001010
000011001011
000100001100
000101001101
000110001110
000111001111
000010010001
000100010010
000110010011
001000010100
001100010110
001100010110
001110010111
000011011001
000110011010
001001011011
001100011100
001111011101
010010011110
010101011111
000100100001
001000100010
001100100011
010000100100
010100100101
011000100110
011100100111
000101101001
001010101010
001111101011
010100101100
011001101101
011110101110
100011101111
000110110001
001100110010
010010110011
011000110100
011110110101
100100110110
101010110111
000111111001
001110111010
010101111011
011100111100
100011111101
101010111110
```

```
THE UPPER BOUND IS NOW : 33
AN IMPROVED SOLUTION IS:
000001--1--1
000100--1100
000010-1--01
000100-1-010
001000-1-100
001007011011
010007-11111
0010001--010
0100001--100
1000071-1111
01000011-011
10000011-11-
1000071111-1
000010-01-1-
0001070-11-1
000010-10--1
000100-10-10
0100001-010-
00100010-01-
01000010-1-0
000010--1-10
001070-101-1
01000001111-
0001071-10-1
00010001 0-1-
0001001-0-01
017007101101
07100010011-
000100-011-0
00100001-10-
0010701-1-10
0010000101--
000100100--1
```

```
EXAMPLE 5 : HOLLERITH CODE TO ASCII (NO PARITY BIT)
                 12 INPUT VARIABLES
                 7 OUTPUT FUNCTIONS


THE SORTED PRODUCT TERMS ARE:
1011111001000010010
1111011101000000000
1111101011000000000
0100000000000000000
0100001100000000110
0100010000000000110
0100011000001000010
0100100010001000010
0100101001000100010
0100110100000000000
0100111000000010010
0101000100000010010
0101001010000010010
0101010010000100010
0101011100000001010
0101100001001000010
0101101010000000000
0101110100001000010
0101111001100000000
0110000001000000000
0110001000100000000
0110010000010000000
0110011000001000000
0110100000000100000
0110101000000010000
0110110000000001000
0110111000000000100
0111000000000000010
0111001000000000001
0111010000010000010
0111011010000001010
0111100100000100010
0111101000000001010
0111110001000001010
0111111001000000110
1000000000000100010
1000001100100000000
1000010100010000000
1000011100001000000
1000100100000100000
1000101100000010000
1000110100000001000
1000111100000000100
1001000100000000010
1001001100000000001
1001010010100000000
1001011010010000000
1001100010001000000
```

```
1001101010000100000
1001110010000010000
1001111010000001000
1010000010000000100
1010001010000000010
1010010010000200001
1010011001010000000
1010100001001000000
1010101001000100000
1010110001000010000
1010111001000001000
1011000001000000100
1011001001000000010
1011010001000000001
1011011100010000010
1011100001010000010
1011101010010000010
1011110010000000110
THE UPPER BOUND IS: 66
THE LIST OBTAINED AT THE END OF PHASE1
7071770010000000110
1000700100000100000
1000707100000010000
1000770100000001000
1000777100000000100
0710007000100000000
1007007100000000001
1007070010100000000
1777077101000000000
1007700010001000000
1007707010000100000
1007770010000010000
1007777010000001000
0717700100000100010
0701777001100000000
1070070010000000001
1000000000000100010
1070700001001000000
1070707001000100000
1000007100100000000
1070777001000001000
1077000001000000100
1777707011000000000
1077070001000000001
1000077100001000000
7077100001010000010
7077107010010000010
0701700001003000010
0777017010000001030
0700001100000000130
1000070100010000020
1007000100020000010
1007077010010000020
```

```
10800   10700000100000000120
10900   07017701000030000010
11000   10700770010100000020
11100   10707700010000010020
11200   07001000100010000030
11300   70170773000100000010
11400   07010001000000030010
11500   07100770020010000000
11600   07107000020001000000
11700   07107070020000010000
11800   07010070100000010030
11900   07010700100001000030
12000   07001070010001000030
12100   07010773000000003010
12200   07170070220000000001
12300   07717770010000000330
12400   10700070100200000210
12500   10770070010200020010
12600   07000172000001000030
12700   07001770020000010030
12800   07100700020100000020
12900   07107700020000001020
13000   07107770220000000100
13100   07000100220000000130
13200   07001701000000222200
13300   07170000220200002010
13400   07017070122222222000
13500   07100002210222222222
13600   01000002222222222222
13700   THE LOWER BOUND IS : 62
```

```
EXAMPLE 6 : FAST SHIFT/ROTATE DECODER
            14 INPUT VARIABLES
            7 OUTPUT FUNCTIONS
THE SORTED PRODUCT TERMS ARE:
0010000011101110000000
0000001001001010000000
0000001011111110000000
000001000-001110000000
000100000-111100001000
000100000-111000000100
000100000-110100000010
000100000-110000000001
000100000-010110000000
000100000-011001000000
000100000-011100100000
000001000-010001000000
000001000-010100100000
000001000-011000010000
000001000-011100001000
000000100-001101000000
000000100-010000100000
000000100-010100010000
000000100-011000001000
001000000-111100010000
001000000-111000001000
001000000-110100000100
001000000-110000000010
001000000-101100000001
001000000-011010000000
001000000-011101000000
000000100-011100000100
000000010-001001000000
000000010-001100100000
000000010-010000010000
00001000111l-11000000
000010000-111100000100
000010000-111000000010
000010000-110100000001
000010000-010010000000
010000000-111100100000
010000000-111000010000
010000000-110100001000
010000000-110000000100
010000000-101100000010
010000000-101000000001
010000000-011110000000
000010000-010101000000
000010000-011000100000
000010000-011100010000
000000010-010100001000
000000010-011000000100
000000010-011100000010
```

```
05500     000000010-000110000000
05600     000000100-111100000001
05700     0000110011111-10000000
05800     100000000-111101000000
05900     100000000-111000100000
06000     100000000-110100010000
06100     100000000-110000001000
06200     100000000-101100000100
06300     100000000-101000000010
06400     100000000-100100000001
06500     000001000-111100000010
06600     000001000-111000000001
06700     00000010--110000100000
06800     00000010--101100010000
06900     00000010--101000001000
07000     00000010--100100000100
07100     00000001--111001000000
07200     00000010--000100000001
07300     00000100--110110000000
07400     00000100--110001000000
07500     00000100--101100100000
07600     00000100--101000010000
07700     00000100--100100001000
07800     00000001--110100100000
07900     00000100--000100000010
08000     00000100--001000000001
08100     00001000--110010000000
08200     00001000--101101000000
08300     00001000--101000100000
08400     00001000--100100010000
08500     00000001--110000010000
08600     00001000--000100000100
08700     00001000--001100000001
08800     00001000--001000000010
08900     00010000--101110000000
09000     00010000--101001000000
09100     00010000--100100100000
09200     00000001--101100001000
09300     00010000--000100001000
09400     00010000--001000000100
09500     00010000--001100000010
09600     00010000--010000000001
09700     011100001111--10000000
09800     00100000--101010000000
09900     00100000--100101000000
10000     00000001--101000000100
10100     00100000--000100010000
10200     00100000--001000001000
10300     00100000--001100000100
10400     00100000--010000000010
10500     00100000--010100000001
10600     01000000--100110000000
10700     00000001--100100000010
```

```
01000000--000100100000
01000000--001000010000
01000000--001100001000
01000000--010000000100
01000000--010100000010
01000000--011000000001
0100000011l-1-10000000
00000001--111110000000
10000000--000101000000
10000000--001000100000
10000000--001100010000
10000000--010000001000
10000000--010100000100
10000000--011000000010
10000000--011100000001
00000010--111010000000
00000010--110101000000
01000000---00001000000
10000000---00010000000
00000001---00000000001
00000010---00000000010
00000100---00000000100
00001000---00000001000
00010000---00000010000
00100000---00000100000
1000000011----10000000
THE UPPER BOUND IS: 127


THE UPPER BOUND IS NOW: 124
AN IMPROVED SOLUTION IS:
0000001001001010000000
000100000-110100000010
000100000-010110000000
000100000-011001000000
000100000-011100100000
000001000-011100001000
000000100-001101000000
000000100-010100010000
000000100-011000001000
000000100-011100000100
000000010-001100100000
000000010-010100001000
000000010-011000000100
000000010-011100000010
001000000-111100010000
001000000-111000001000
001000000-110100000100
001000000-101100000001
001000000-011010000000
001000000-011101000000
000000010-000110000000
000010000-111100000100
000010000-111000000010
```

```
16200     000010000-110100000001
16300     000010000-010101000000
16400     000010000-011000100000
16500     000010000-011100010000
16600     000000010-001001000000
16700     010000000-111100100000
16800     010000000-111000010000
16900     010000000-110100001000
17000     010000000-101100000010
17100     010000000-101000000001
17200     010000000-011110000000
17300     7777771011111-10000000
17400     000000100-111100000001
17500     000001000-111100000010
17600     000001000-111000000001
17700     000001000-001110000000
17800     000001000-010100100000
17900     000001000-011000010000
18000     000100000-111100001000
18100     000100000-111000000100
18200     100000000-111101000000
18300     100000000-111000100000
18400     100000000-110100010000
18500     100000000-101100000100
18600     100000000-101000000010
18700     100000000-100100000001
18800     10000000--011100000001
18900     00000100--110110000000
19000     00000001--101100001000
19100     00000100--101100100000
19200     00000100--101000010000
19300     00000100--100100001000
19400     00000001--101000000100
19500     00000100--000100000010
19600     00000100--001000000001
19700     00000001--100100000010
19800     00001000--101101000000
19900     00001000--101000100000
20000     00001000--100100010000
20100     00000001--111001000000
20200     00001000--000100000100
20300     00001000--001100000001
20400     00001000--001000000010
20500     777710001111--10000000
20600     00010000--101110000000
20700     00010000--101001000000
```

```
20800    00010000--100100100000
20900    00000010--111010000000
21000    00010000--000100001000
21100    00010000--001000000100
21200    00010000--001100000010
21300    00000010--110101000000
21400    00100000--101010000000
21500    00100000--100101000000
21600    00000001--110100100000
21700    00100000--000100010000
21800    00100000--001000001000
21900    00100000--001100000100
22000    00000010--101100010000
22100    00100000--010100000001
22200    77100000111-1-10000000
22300    01000000--100110000000
22400    00000010--101000001000
22500    01000000--000100100000
22600    01000000--001000010000
22700    01000000--001100001000
22800    00000010--100100000100
22900    01000000--010100000010
23000    01000000--011000000001
23100    00000001--111110000000
23200    10000000--000101000000
23300    10000000--001000100000
23400    10000000--001100010000
23500    00000010--000100000001
23600    10000000--010100000100
23700    10000000--011000000010
23800    00000010---00000000010
23900    00000100--110001000000
24000    00000100---00000000100
24100    00001000--110010000000
24200    00001000---00000001000
24300    00010000---00000010000
24400    00010000--010000000001
24500    00100000---00000100000
24600    00100000--010000000010
24700    01000000---00001000000
24800    01000000--010000000100
24900    10000000---00010000000
25000    10000000--010000001000
25100    00000001--110000010000
25200    00000001---00000000001
25300    00000010--110000100000
25400    1000000011----10000000
25500    000001000--10001000000
25600    000010000--10010000000
25700    000100000--10000000001
25800    001000000--10000000010
25900    010000000--10000000100
26000    100000000--10000001000
26100    000000010--10000010000
26200    000000100--10000100000
```

```
EXAMPLE 7 : SPECIAL COUNTER
            5 INPUT VARIABLES
            5 OUTPUT FUNCTIONS

THE SORTED PRODUCT TERMS ARE:
1111111110
0000100000
0001000001
0001100010
0010000011
0010100100
0011000101
0011100110
0100000111
1000001000
1000110000
0100110001
0101001001
1001001010
1001110010
0101110011
0110001011
1010001100
1010110100
0110110101
0111001101
1011001110
1011110110
0111110111
1100001111
1100111000
1101011001
1101111010
1110011011
1110111100
1111011101
THE UPPER BOUND IS: 31

THE UPPER BOUND IS NOW: 18
AN IMPROVED SOLUTION IS:
07100-1011
100000111-
001000-011
000100--01
0001710-1-
01070-1-01
7100711--0
00010---10
00001-0--0
10000-1--0
700071---0
00100--1-0
010000-111
1700011-0-
0700110--1
00100--10-
00107101--
17000110--
```

```
EXAMPLE 8 : F(W,X,Y,Z) = (W'X'Y'Z' + WXYZ)
               4 INPUT VARIABLES
               1 OUTPUT FUNCTION

THE SORTED PRODUCT TERMS ARE:
11110
10001
10010
10011
10100
10101
10110
10111
11000
11001
11010
11011
11100
11101
THE UPPER BOUND IS: 14
THE OPTIMAL REALIZATION IS:
11--0
1--01
1-01-
101--
```

```
EXAMPLE 9 : SPECIAL DECODER
            8 INPUT VARIABLES
            1 OUTPUT FUNCTION


THE SORTED PRODUCT TERMS ARE:
110010101
100001000
100001001
100010000
100010001
100010010
100010011
100010100
100010101
100100100
100100101
100100110
100100111
100101000
100101001
100110000
100110001
101000000
101000001
101000010
101000011
101000100
101000101
101000110
101000111
101010110
101010111
101011000
101011001
101100000
101100001
101100010
101100011
101110010
101110011
101110100
101110101
101110110
101110111
101111000
101111001
110001000
110001001
110010000
110010001
110010010
110010011
110010100
THE UPPER BOUND IS: 48
```

```
THE UPPER BOUND IS NOW: 12
AN IMPROVED SOLUTION IS:
1011101--
101-1100-
1001001--
1011-001-
1-000100-
101-1011-
101-000--
1-0010-0-
100-0100-
101000---
100-1000-
1-00100--
```

```
EXAMPLE 10 : SPECIAL FUNCTION FUN
             5 INPUT VARIABLES
             1 OUTPUT FUNCTION


THE SORTED PRODUCT TERMS ARE:
111011
100000
100001
100010
100011
100100
100101
101100
101000
110011
110101
110111
111100
111101
111111
111110
THE UPPER BOUND IS: 16



THE OPTIMAL REALIZATION IS:
1-0101
10--00
1111--
11--11
1000--
```

Appendix C

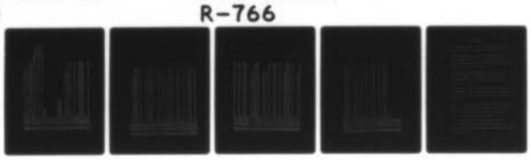
Detailed listing of output

from "MINI" for test problem 1

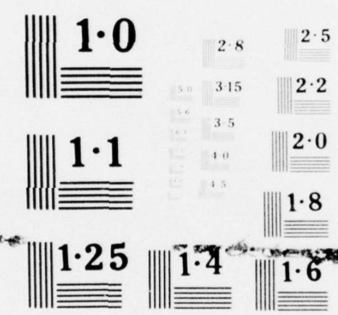
```
EXAMPLE 1 : 7 SEGMENT DECODER
            4 INPUT VARIABLES
            7 OUTPUT FUNCTIONS

THE INPUT PRODUCT TERMS ARE:
1000000-00
100000100-
1000000 10-
10000001-0
2100000-000
21000000-10
2210000001-1
2210000-000
2210000100-
22100000001-
22210000010-
22210000100-
2221000001-
2221000001-10
222210000101
22221000-000
22221000001-
222210000-10
2222100010--
22222100111
222222100100
22222210-00-
222222210--1
2222222211--
22222221-00-
999
THE SORTED PRODUCT TERMS ARE:
22222120102
22222 22121
22222122111
2210000001-1
21111000021-
1000000-00
1011000010-
1001000010-
10000011-0
211100-000
2101100-10
2100001--1
2000001 1--
20000120--
200011-00-
THE UPPER BOUND IS: 15
THE LIST OBTAINED AT THE END OF PHASE1
00717770013
17077070130
7077077100-
10777070123
17707772220
01777700230
00177770331
THE LOWER BOUND IS : 7
STACK 1 IS:
00717770217 17077070130 7077077100- 10777070103 17707772200 01777700230 00100770331
STACK 2 IS:
```

AD-A043 361 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORD SCIENCE LAB F/G 9/2
AN ALGORITHM FOR MINIMIZING PROGRAM LOGIC ARRAY REAL--ETC(U)
APR 77 A.G. SOONG DAAB07-72-C-0259
UNCLASSIFIED R-766 NL

2 OF 2
AD
A043361



END
DATE
FILMED
1-78
DDC



NATIONAL BUREAU OF STANDARDS
MICROCOPY RESOLUTION TEST CHART

```
06100   0071770221- 17077070170 7077077100- 10777070103 17707772200 01777700230 00100770331
06200   STACK 3 IS:
06300   0071770221- 100700701-0 7077077100- 1077707010Y 17707772200 01777700230 00100770331
06400   STACK 4 IS:
06500   0071770221- 100700701-0 7077077100- 1007007010- 1770777X200 01777700230 00100770331 70771070101
06600   STACK 5 IS:
06700   0071770221- 100700701-0 7077077100- 1007007010- 1770777-000 01777700X30 00100770331 70771070101
06800   STACK 6 IS:
06900   0071770221- 100700701-0 7077077100- 1007007010- 1770777-000 01077000-10 00100770Y31 70771070101 70070170300
07000   STACK 7 IS:
07100   0071770221- 100700701-0 7077077100- 1007007010- 1770777-000 01077000-10 00100770-11 70771070101 700701707Y00 00000172023
07200
07300   STACK 8 IS:
07400   0071770221- 100700701-0 7077077100- 1007007010- 1770777-000 01077000-10 00100770-11 70771070101 70000170-00 0000017X023
07500
07600   STACK 9 IS:
07700   0071770221- 100700701-0 7077077100- 1007007010- 1770777-000 01077000-10 00100770-11 70771070101 70000170-00 0000017-00Y
07800
07900   THE UPPER BOUND IS NOW: 9
08000   AN IMPROVED SOLUTION IS:
08100   0071770221-
08200   100700701-0
08300   7077077100-
08400   1770777-000
08500   01077000-10
08600   00100770-11
08700   70771070101
08800   70000170-00
08900   0000017-00-
09000   BACK-TRACKING
09100   BACK-TRACKING: GATCON[STAGE] GEQ UPBOUND
09200   BACK-TRACKING: GATCON[STAGE] GEQ UPBOUND
09300   BACK-TRACKING: GATCON[STAGE] GEQ UPBOUND
09400   BACK-TRACKING: GATCON[STAGE] GEQ UPBOUND
09500   BACK-TRACKING
09600   BACK-TRACKING
09700   STACK 3 IS:
09800   0071770221- 100700701-0 7077077100- 10777070101 1770777X200 01777700230 00100770331
09900   STACK 4 IS:
10000   0071770221- 100700701-0 7077077100- 10777070101 1770777-000 01777700X30 00100770331
10100   STACK 5 IS:
10200   0071770221- 100700701-0 7077077100- 10777070101 1770777-000 01077000-10 00100770Y31 70070170300
10300   BACK-TRACKING
10400   BACK-TRACKING
10500   STACK 4 IS:
10600   0071770221- 100700701-0 7077077100- 10777070101 1770777-000 01777000Y0 00100770331 71077070310
10700   BACK-TRACKING
10800   BACK-TRACKING
10900   STACK 3 IS:
11000   0071770221- 100700701-0 7077077100- 10777070101 17707770X00 01777700230 00100770331 71777773000
11100   STACK 4 IS:
11200   0071770221- 100700701-0 7077077100- 10777070101 10000770-00 01777700X30 00100770331 71777773000
11300   STACK 5 IS:
11400   0071770221- 100700701-0 7077077100- 10777070101 10000770-00 01077000-10 00100770Y31 71777773000
11500   BACK-TRACKING
11600   BACK-TRACKING
11700   BACK-TRACKING
11800   STACK 3 IS:
11900   0071770221- 100700701-0 7077077100- 10777070101 17707770000 01777700X30 00100770331 71777773000
12000   BACK-TRACKING
```

```
12100   BACK-TRACKING
12200   STACK 2 IS:
12300   0071770001= 17077070110 7077077100= 1077707010Y 17707770000 01777700030 00100770331
12400   STACK 3 IS:
12500   0071770001= 17077070110 7077077100= 1007007010= 1770777x000 01777700030 00100770331 70771070101
12600   STACK 4 IS:
12700   0071770001= 17077070110 7077077100= 1007007010= 1770777=000 01777700x30 00100770331 70771070101
12800   BACK-TRACKING
12900   STACK 4 IS:
13000   0071770001= 17077070110 7077077100= 1007007010= 1770777=000 017777000Y0 00100770331 70771070101
13100   BACK-TRACKING
13200   BACK-TRACKING
13300   BACK-TRACKING
13400   STACK 2 IS:
13500   0071770001= 17077070110 7077077100= 10777070101 1770777x000 01777700030 00100770331
13600   STACK 3 IS:
13700   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01777700x30 00100770331 10070770300
13800   STACK 4 IS:
13900   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01077000=10 00100770Y31 10070770300
14000   STACK 5 IS:
14100   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01077000=10 00100770=11 10070770Y00
14200   BACK-TRACKING
14300   BACK-TRACKING
14400   STACK 4 IS:
14500   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01077000=10 001007701Y1 10070770300
14600   STACK 5 IS:
14700   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01077000=10 001000701=1 10070770Y00
14800   BACK-TRACKING
14900   BACK-TRACKING
15000   STACK 4 IS:
15100   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01077000=10 00100770111 10070770Y00
15200   BACK-TRACKING
15300   BACK-TRACKING
15400   STACK 3 IS:
15500   0071770001= 17077070110 7077077100= 10777070101 1770777=000 017777000Y0 00100770331 10070770300
15600   STACK 4 IS:
15700   0071770001= 17077070110 7077077100= 10777070101 1770777=000 017077000=0 00100770Y31 10070770300
15800   STACK 5 IS:
15900   0071770001= 17077070110 7077077100= 10777070101 1770777=000 017077000=0 00100770=11 10070770Y00
16000   BACK-TRACKING
16100   BACK-TRACKING
16200   STACK 4 IS:
16300   0071770001= 17077070110 7077077100= 10777070101 1770777=000 017077000=0 001007701Y1 10070770300
16400   STACK 5 IS:
16500   0071770001= 17077070110 7077077100= 10777070101 1770777=000 017077000=0 001000701=1 10070770Y00
16600   BACK-TRACKING
16700   BACK-TRACKING
16800   STACK 4 IS:
16900   0071770001= 17077070110 7077077100= 10777070101 1770777=000 017077000=0 00100770111 10070770Y00
17000   BACK-TRACKING
17100   BACK-TRACKING
17200   STACK 3 IS:
17300   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01777700010 00100770Y31 10070770300
17400   STACK 4 IS:
17500   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01777700010 00100770=11 10070770Y00
17600   BACK-TRACKING
17700   BACK-TRACKING
17800   STACK 3 IS:
17900   0071770001= 17077070110 7077077100= 10777070101 1770777=000 01777700010 001007701Y1 10070770300
18000   STACK 4 IS:
```

```
18100   0271770001- 1707707011 0 7077077100- 10777070101 1770777-000 01777700010 001000701-1 10070770Y22
18200   BACK-TRACKING
18300   BACK-TRACKING
18400   STACK 3 IS:
18500   0271770021- 17077070110 7077077100- 10777070101 1770777-000 01777700010 00100770111 10070770Y22
18600   BACK-TRACKING
18700   BACK-TRACKING
18800   STACK 2 IS:
18900   0271770021- 17077070110 7077077100- 10777070101 1770777 0X00 01777700230 00100770331 71777773000
19000   STACK 3 IS:
19100   0271770001- 17077070110 7077077100- 10777070101 10000770-00 01777700X30 00100770331 71777773000
19200   BACK-TRACKING
19300   BACK-TRACKING
19400   BACK-TRACKING
19500   STACK 1 IS:
19600   02717770011 170770701Y0 7077077100- 10777070103 17707772200 01777700230 00100770331
19700   STACK 2 IS:
19800   02717770011 100700701-0 7077077100- 1077707010Y 17707772200 01777700230 00100770331
19900   STACK 3 IS:
20000   02717770011 100700701-0 7077077100- 1007007010- 1770777X200 01777700230 00100770331 70771070101
20100   STACK 4 IS:
20200   02717770011 100700701-0 7077077100- 1007007010- 1770777-000 01777700X30 00100770331 70771070101
20300   BACK-TRACKING
20400   BACK-TRACKING
20500   BACK-TRACKING
20600   STACK 2 IS:
20700   02717770011 100700701-0 7077077100- 10777070101 1770777X200 01777700230 00100770331
20800   STACK 3 IS:
20900   02717770011 100700701-0 7077077100- 10777070101 1770777-000 01777700X30 00100770331
21000   BACK-TRACKING
21100   STACK 3 IS:
21200   02717770011 100700701-0 7077077100- 10777070101 1770777-000 01777700Y0 00100770331 71077070310
21300   BACK-TRACKING
21400   BACK-TRACKING
21500   STACK 2 IS:
21600   02717770011 100700701-0 7077077100- 10777070101 17707770X00 01777700230 00100770331 71777773000
21700   STACK 3 IS:
21800   02717770011 100700701-0 7077077100- 10777070101 10000770-00 01777700X30 00100770331 71777773000
21900   BACK-TRACKING
22000   BACK-TRACKING
22100   STACK 2 IS:
22200   02717770011 100700701-0 7077077100- 10777070101 17707770000 01777700X30 00100770331 71777773000
22300   BACK-TRACKING
22400   BACK-TRACKING
22500   STACK 1 IS:
22600   02717770011 17077070110 7077077100- 1077707010Y 17707772200 01777700230 00100770331
22700   STACK 2 IS:
22800   02717770011 17077070110 7077077100- 1007007010- 1770777X200 01777700230 00100770331 70771070101
22900   STACK 3 IS:
23000   02717770011 17077070110 7077077100- 1007007010- 1770777-000 01777700X30 00100770331 70771070101
23100   BACK-TRACKING
23200   STACK 3 IS:
23300   02717770011 17077070110 7077077100- 1007007010- 1770777-000 01777700Y0 00100770331 70771070101
23400   BACK-TRACKING
23500   BACK-TRACKING
23600   BACK-TRACKING
23700   STACK 1 IS:
23800   02717770011 17077070110 7077077100- 10777070101 1770777X200 01777700230 00100770331
23900   STACK 2 IS:
24000   02717770011 17077070110 7077077100- 10777070101 1770777-000 01777700X30 00100770331 10070770322
```

```
24100   BACK-TRACKING
24200   STACK 2 IS:
24300   02717777011 17077070110 7077077100= 10777070101 1770777=000 01777700010 00100770331 10070770322
24400   STACK 3 IS:
24500   02717777011 17077070110 7077077100= 10777070101 1770777=000 0170770000=0 00100770Y31 10070770322
24600   STACK 4 IS:
24700   02717777011 17077070110 7077077100= 10777070101 1770777=000 0170770000=0 00100770=11 10070770Y22
24800   BACK-TRACKING
24900   BACK-TRACKING
25000   STACK 3 IS:
25100   02717777011 17077070110 7077077100= 10777070101 1770777=000 0170770000=0 0010077001Y1 10070770322
25200   STACK 4 IS:
25300   02717777011 17077070110 7077077100= 10777070101 1770777=000 0170770000=0 001000701=1 10070770Y22
25400   BACK-TRACKING
25500   BACK-TRACKING
25600   STACK 3 IS:
25700   02717777011 17077070110 7077077100= 10777070101 1770777=000 0170770000=0 00100770111 10070770Y20
25800   BACK-TRACKING
25900   BACK-TRACKING
26000   STACK 2 IS:
26100   02717777011 17077070110 7077077100= 10777070101 1770777=000 01777700010 00100770Y31 10070770322
26200   STACK 3 IS:
26300   02717777011 17077070110 7077077100= 10777070101 1770777=000 01777700010 00100770=11 10070770Y22
26400   BACK-TRACKING
26500   BACK-TRACKING
26600   STACK 2 IS:
26700   02717777011 17077070110 7077077100= 10777070101 1770777=000 01777700010 0010077001Y1 10070770322
26800   STACK 3 IS:
26900   02717777011 17077070110 7077077100= 10777070101 1770777=000 01777700010 001000701=1 10070770Y22
27000   BACK-TRACKING
27100   BACK-TRACKING
27200   STACK 2 IS:
27300   02717777011 17077070110 7077077100= 10777070101 1770777=000 01777700010 00100770111 10070770Y22
27400   BACK-TRACKING
27500   BACK-TRACKING
27600   STACK 1 IS:
27700   02717777011 17077070110 7077077100= 10777070101 17707770X00 01777700230 00100770331 71777773000
27800   STACK 2 IS:
27900   02717777011 17077070110 7077077100= 10777070101 10000770=00 01777700X30 00100770331 71777773000
28000   BACK-TRACKING
28100   BACK-TRACKING
28200   BACK-TRACKING
28300   THE OPTIMAL REALIZATION IS:
28400   0271770001=
28500   100700701=0
28600   7077077100=
28700   1770777=000
28800   01077000=10
28900   00100770=11
29000   70771070101
29100   70000170=00
29200   0000017=00=
```

REFERENCES

1. E. J. McCluskey, Introduction to the Theory of Switching Circuits. New York : McGraw Hill Book Company, 1965.

2. E. S. Davidson, "An algorithm for NAND decomposition of combinational systems," Ph.D dissertation, Department of Electrical Engineering and Coordinated Science Laboratory, University of Illinois, 1968.

3. S. Muroga and T. Nakagawa, "Exposition of Davidson's Thesis 'An algorithm for NAND decomposition of combinational Switching systems'," Department of Computer Science, University of Illinois.

4. T. Nakagawa, "A branch-and-bound algorithm for optimal AND-OR networks (the algorithm description)," Report No. 462, Department of Computer Science, University of Illinois, June 1971.

5. H. C. Lai, S. Muroga and T. Nakagawa, "Pruning and branching methods for designing optimal networks by the branch-and-bound method," Report No. 471, Department of Computer Science, University of Illinois, 1971.

6. T. Nakagawa, "Reference manual of FORTRAN program ILLOD-(AND-OR-B) for optimal AND-OR networks," Report No. 488, Department of Computer Science, University of Illinois, December 1971.